Allocation Removal by Partial Evaluation in a Tracing JIT

Carl Friedrich Bolz[a]   Antonio Cuni[a]   Maciej Fijałkowski[b]   Michael Leuschel[a]

Samuele Pedroni[c]   Armin Rigo[a]

[a]Heinrich-Heine-Universität Düsseldorf, STUPS Group, Germany
[b]merlinux GmbH, Hildesheim, Germany
[c]Open End, Göteborg, Sweden

cfbolz@gmx.de   anto.cuni@gmail.com   fijal@merlinux.eu   leuschel@cs.uni-duesseldorf.de
samuele.pedroni@gmail.com   arigo@tunes.org

Abstract

The performance of many dynamic language implementations suffers from high allocation rates and runtime type checks. This makes dynamic languages less applicable to purely algorithmic problems, despite their growing popularity. In this paper we present a simple compiler optimization based on online partial evaluation to remove object allocations and runtime type checks in the context of a tracing JIT. We evaluate the optimization using a Python VM and find that it gives good results for all our (real-life) benchmarks. [1]

Categories and Subject Descriptors   D.3.4 [*Programming Languages*]: Processors—code generation, interpreters, run-time environments

General Terms   Languages, Performance, Experimentation

Keywords   Tracing JIT, Partial Evaluation, Optimization

1.   Introduction

The objective of a just-in-time (JIT) compiler for a dynamic language is to improve the speed of the language over an implementation of the language that uses interpretation. The first goal of a JIT is therefore to remove the interpretation overhead, i.e. the overhead of bytecode (or AST) dispatch and the overhead of the interpreter's data structures, such as operand stack etc. The second important problem that any JIT for a dynamic language needs to solve is how to deal with the overhead of boxing primitive types and of type dispatching. Those are problems that are usually not present or at least less severe in statically typed languages.

Boxing of primitive types is necessary because dynamic languages need to be able to handle all objects, even integers, floats, booleans etc. in the same way as user-defined instances. Thus those primitive types are usually *boxed*, i.e., a small heap-structure is allocated for them that contains the actual value. Boxing primitive types can be very costly, because a lot of common operations, particularly all arithmetic operations, have to produce new boxes, in addition to the actual computation they do. Because the boxes are allocated on the heap, producing many of them puts pressure on the garbage collector.

Type dispatching is the process of finding the concrete implementation that is applicable to the objects at hand when performing a generic operation on them. An example would be the addition of two objects: For addition the types of the concrete objects need to be checked and the suiting implementation chosen. Type dispatching is a very common operation in modern[2] dynamic languages because no types are known at compile time. Therefore all operations need it.

A recently popular approach to implementing just-in-time compilers for dynamic languages is that of a tracing JIT. A tracing JIT works by observing the running program and recording its hot spots into *linear execution traces*. Those traces are optimized and turned into machine code.

One reason for the popularity of tracing JITs is their relative simplicity. They can often be added to an existing interpreter, reusing a lot of the interpreter's infrastructure. They give some important optimizations like inlining and constant-folding for free. A tracing JIT always produces linear pieces of code, which simplifies many of the hard algorithms in a compiler, such as register allocation.

The use of a tracing JIT can remove the overhead of bytecode dispatch and that of the interpreter data structures. In this paper we want to present a new optimization that can be added to a tracing JIT that further removes some of the overhead more closely associated to dynamic languages, such as boxing overhead and type dispatching. Our experimental platform is the PyPy project, which is an environment for implementing dynamic programming languages. PyPy and tracing JITs are described in more detail in Section 2. Section 3 analyzes the problem to be solved more closely.

The core of our trace optimization technique can be viewed as partial evaluation: the partial evaluation performs a form of escape analysis [4] on the traces and makes some objects that are allocated in the trace *static,* which means that they do not occur any more in the optimized trace. This technique is informally described in Section 4; a more formal description is given in Section 5. The introduced techniques are evaluated in Section 6 using PyPy's Python interpreter.

The contributions made by this paper are:

---

[1] This research is partially supported by the BMBF funded project PyJIT (nr. 01QE0913B; Eureka Eurostars).

[2] For languages in the LISP family, basic arithmetic operations are typically not overloaded; even in Smalltalk, type dispatching is much simpler than in Python or JavaScript.

2010/10/22

1. A description of an efficient and effective algorithm for removing object allocations in a tracing JIT.

2. A characterization of this algorithm as partial evaluation.

3. Performance benchmarks for this algorithm.

2. Background

2.1 PyPy

The work described in this paper was done in the context of the PyPy project[3] [22]. PyPy is an environment where dynamic languages can be implemented in a simple yet efficient way. When implementing a language with PyPy one writes an *interpreter* for the language in *RPython* [1]. RPython ("restricted Python") is a subset of Python chosen in such a way that type inference becomes possible. The language interpreter can then be compiled ("translated") with PyPy's tools into a VM on C level. During translation to C, many low-level aspects of the final VM, such as object layout, garbage collection and memory model, are woven into the generated code. Therefore the interpreter itself can remain at a relatively high level of abstraction.

A number of languages have been implemented with PyPy. The project was initiated to get a better Python implementation, which inspired the name of the project and is still the main focus of development. In addition a number of other languages were implemented, among them a Prolog interpreter [7], a Smalltalk VM [6] and a GameBoy emulator [8].

The feature that makes PyPy more than a compiler with a runtime system is its support for automated JIT compiler generation [5]. During the translation to C, PyPy's tools can generate a tracing just-in-time compiler for the language that the interpreter is implementing. This process is mostly automatic; it only needs to be guided by the language implementer using a small number of source-code hints in the interpreter. Mostly-automatically generating a JIT compiler has many advantages over writing one manually, an error-prone and tedious process. By construction, the generated JIT has the same semantics as the interpreter. Optimizations can be shared between different languages implemented with PyPy.

Moreover, thanks to the internal design of the JIT generator, it is very easy to add new *backends* for producing the actual machine code. Examples of JIT backends that are implemented are those for Intel x86 and x86-64 and an experimental one for the CLI .NET Virtual Machine [12].

2.2 Tracing JIT Compilers

Tracing JITs are a recently popular approach to write just-in-time compilers for dynamic languages. Their origins lie in the Dynamo project, which used a tracing approach to optimize machine code using execution traces [2]. Tracing JITs have then be adapted to be used for a very light-weight Java VM [15] and afterwards used in several implementations of dynamic languages, such as JavaScript [13], Lua[4] and now Python (and other languages) via PyPy.

The core idea of tracing JITs is to focus the optimization effort of the JIT compiler on the commonly executed, i.e., *hot* paths of the core loops of the program and to just use an interpreter for the less commonly executed parts. VMs that use a tracing JIT are mostly mixed-mode execution environments, they contain both an interpreter and a JIT compiler. By default the interpreter is used to execute the program, doing some light-weight profiling at the same time. This profiling is used to identify the hot loops of the program. If a hot loop is found in that way, the interpreter enters a special *tracing mode*. In this tracing mode, the interpreter tries to

---

[3] http://pypy.org

[4] http://luajit.org/

record all operations that it is executing while running one iteration of the hot loop. This history of executed operations of one loop is called a *trace*. Because the trace corresponds to one iteration of a loop, it always ends with a jump to its own beginning. The trace also contains all operations that are performed in functions that were called in the loop, thus a tracing JIT automatically performs inlining. This trace of operations subsequently forms the basis of the generated code. The trace is first optimized, and then turned into machine code. Both optimization and machine code generation are simple, because the traces are linear. This linearity makes many optimizations a lot more tractable, and the inlining that happens gives the optimizations automatically more context to work with.

Since the trace corresponds to one concrete execution of a loop, the code generated from it is only one possible path through the loop. To make sure that the trace maintains the correct semantics, it contains a *guard* at all places where the execution could have diverged from the path. Those guards check the assumptions under which execution can stay on the trace. As an example, if a loop contains an if-statement, the trace will contain the execution of one of the paths only, which is the path that was taken during the production of the trace. The trace will also contain a guard that checks that the condition of the if-statement is the same as during tracing, because if it isn't, the rest of the trace would not be valid.

When generating machine code, every guard is turned into a quick check to see whether the assumption still holds. When such a guard is hit during the execution of the machine code and the assumption does not hold, the execution of the machine code is stopped, and interpreter continues to run from that point on. These guards are the only mechanism to stop the execution of a trace, the loop end condition also takes the form of a guard.

If one specific guard fails often enough, the tracing JIT will generate a new trace that starts exactly at the position of the failing guard [14]. The existing assembler is patched to jump to the new trace when the guard fails. This approach guarantees that all the hot paths in the program will eventually be traced and compiled into efficient code.

2.3 Running Example

For the purpose of this paper, we are going to use a tiny interpreter for a dynamic language with a very simple object model, that just supports an integer and a float type. The objects support only two operations, add, which adds two objects (promoting ints to floats in a mixed addition) and is_positive, which returns whether the number is greater than zero. The implementation of add uses classical Smalltalk-like double-dispatching. The classes can be seen in Figure 1 (written in RPython).

Using these classes to implement arithmetic shows the basic problem of a dynamic language implementation. All the numbers are instances of either BoxedInteger or BoxedFloat, therefore they consume space on the heap. Performing many arithmetic operations produces lots of garbage quickly, putting pressure on the garbage collector. Using double dispatching to implement the numeric tower needs two method calls per arithmetic operation, which is costly due to the method dispatch.

Let us now consider a simple "interpreter" function f that uses the object model (see the bottom of Figure 1). The loop in f iterates y times, and computes something in the process. Simply running this function is slow, because there are lots of virtual method calls inside the loop, one for each is_positive and even two for each call to add. These method calls need to check the type of the involved objects repeatedly and redundantly. In addition, a lot of objects are created when executing that loop, many of these objects are short-lived. The actual computation that is performed by f is simply a sequence of float or integer additions.

```python
class Base(object):
    pass

class BoxedInteger(Base):
    def __init__(self, intval):
        self.intval = intval

    def add(self, other):
        return other.add__int(self.intval)

    def add__int(self, intother):
        return BoxedInteger(intother + self.intval)

    def add__float(self, floatother):
        floatvalue = floatother + float(self.intval)
        return BoxedFloat(floatvalue)

    def is_positive(self):
        return self.intval > 0

class BoxedFloat(Base):
    def __init__(self, floatval):
        self.floatval = floatval

    def add(self, other):
        return other.add__float(self.floatval)

    def add__int(self, intother):
        floatvalue = float(intother) + self.floatval
        return BoxedFloat(floatvalue)

    def add__float(self, floatother):
        return BoxedFloat(floatother + self.floatval)

    def is_positive(self):
        return self.floatval > 0.0


def f(y):
    res = BoxedInteger(0)
    while y.is_positive():
        res = res.add(y).add(BoxedInteger(-100))
        y = y.add(BoxedInteger(-1))
    return res
```

Figure 1. An "Interpreter" for a Tiny Dynamic Language Written in RPython

If the function is executed using the tracing JIT, with y being a BoxedInteger, the produced trace looks like the one of Figure 2 (lines starting with a hash "#" are comments). The trace corresponds to one iteration of the while-loop in f.

The operations in the trace are indented corresponding to the stack level of the function that contains the traced operation. The trace is in single-assignment form, meaning that each variable is assigned a value exactly once. The arguments p_0 and p_1 of the loop correspond to the live variables y and res in the while-loop of the original function.

The operations in the trace correspond to the operations in the RPython program in Figure 1:

- new creates a new object.

- get reads an attribute of an object.

- set writes to an attribute of an object.

```
# arguments to the trace: p₀, p₁                    1
# inside f: res.add(y)                              2
guard_class(p₁, BoxedInteger)                       3
    # inside BoxedInteger.add                       4
    i₂ = get(p₁, intval)                            5
    guard_class(p₀, BoxedInteger)                   6
        # inside BoxedInteger.add__int              7
        i₃ = get(p₀, intval)                        8
        i₄ = int_add(i₂, i₃)                        9
        p₅ = new(BoxedInteger)                      10
            # inside BoxedInteger.__init__          11
            set(p₅, intval, i₄)                     12
                                                    13
# inside f: BoxedInteger(-100)                      14
p₆ = new(BoxedInteger)                              15
    # inside BoxedInteger.__init__                  16
    set(p₆, intval, -100)                           17
                                                    18
# inside f: .add(BoxedInteger(-100))               19
guard_class(p₅, BoxedInteger)                       20
    # inside BoxedInteger.add                       21
    i₇ = get(p₅, intval)                            22
    guard_class(p₆, BoxedInteger)                   23
        # inside BoxedInteger.add__int              24
        i₈ = get(p₆, intval)                        25
        i₉ = int_add(i₇, i₈)                        26
        p₁₀ = new(BoxedInteger)                     27
            # inside BoxedInteger.__init__          28
            set(p₁₀, intval, i₉)                    29
                                                    30
# inside f: BoxedInteger(-1)                        31
p₁₁ = new(BoxedInteger)                             32
    # inside BoxedInteger.__init__                  33
    set(p₁₁, intval, -1)                            34
                                                    35
# inside f: y.add(BoxedInteger(-1))                36
guard_class(p₀, BoxedInteger)                       37
    # inside BoxedInteger.add                       38
    i₁₂ = get(p₀, intval)                           39
    guard_class(p₁₁, BoxedInteger)                  40
        # inside BoxedInteger.add__int              41
        i₁₃ = get(p₁₁, intval)                      42
        i₁₄ = int_add(i₁₂, i₁₃)                     43
        p₁₅ = new(BoxedInteger)                     44
            # inside BoxedInteger.__init__          45
            set(p₁₅, intval, i₁₄)                   46
                                                    47
# inside f: y.is_positive()                         48
guard_class(p₁₅, BoxedInteger)                      49
    # inside BoxedInteger.is_positive               50
    i₁₆ = get(p₁₅, intval)                          51
    i₁₇ = int_gt(i₁₆, 0)                            52
# inside f                                          53
guard_true(i₁₇)                                     54
jump(p₁₅, p₁₀)                                      55
```
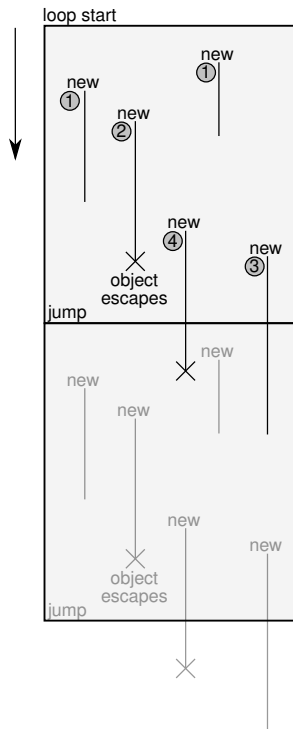
Figure 2. An Unoptimized Trace of the Example Interpreter

Figure 3. Object Lifetimes in a Trace

Figure 3 shows a trace before optimization, together with the lifetime of various kinds of objects created in the trace. It is executed from top to bottom. At the bottom, a jump is used to execute the same loop another time (for clarity, the figure shows two iterations of the loop). The loop is executed until one of the guards in the trace fails, and the execution is aborted and interpretation resumes.

Some of the operations within this trace are `new` operations, which each create a new instance of some class. These instances are used for some time, e.g., by calling methods on them (which are inlined into the trace), reading and writing their fields. Some of these instances *escape*, which means that they are stored in some globally accessible place or are passed into a non-inlined function via a residual call.

Together with the `new` operations, the figure shows the lifetimes of the created objects. The objects that are created within a trace using `new` fall into one of several categories:

1. Objects that live for some time, and are then just not used any more afterwards.

2. Objects that live for some time and then escape.

3. Objects that live for some time, survive across the jump to the beginning of the loop, and are then not used any more.

4. Objects that live for some time, survive across the jump, and then escape. To these we also count the objects that live across several jumps and then either escape or stop being used.

The objects that are allocated in the example trace in Figure 2 fall into categories 1 and 3. Objects stored in p_5, p_6, p_{11} are in category 1, objects in p_{10}, p_{15} are in category 3.

The creation of objects in category 1 is removed by the optimization described in Sections 4 and 5. Objects in the other categories are partially optimized by this approach as well.[6]

4. Allocation Removal in Traces

The main insight to improve the code shown in Section 2.3 is that objects in category 1 do not survive very long – they are used only inside the loop and there is no other outside reference to them. The optimizer identifies objects in category 1 and removes the allocation of these objects, and all operations manipulating them.

This is a process that is usually called *escape analysis* [17]. In this paper we will perform escape analysis by using partial evaluation. The use of partial evaluation is a bit peculiar in that it receives no static input arguments for the trace, but it is only used to optimize operations within this process by examining the example trace in Figure 2. The final trace after optimization can be seen in Figure 4 (the line numbers are the lines of the unoptimized trace where the operation originates).

To optimize the trace, it is traversed from beginning to end and an output trace is produced. Every operation in the input trace is either removed or copied into the output trace. Sometimes new operations need to be produced as well. The optimizer can only remove operations that manipulate objects that have been allocated within the trace, while all other operations are copied to the output trace unchanged.

Looking at the example trace of Figure 2, the operations in lines 1–9 are manipulating objects which existed before the trace and that are passed in as arguments: therefore the optimizer just copies them into the output trace.

The following operations (lines 10–17) are more interesting:

$$p_5 = \texttt{new(BoxedInteger)} \qquad \text{10}$$

- `guard_class` is a precise type check and precedes an (inlined) method call and is followed by the trace of the called method.

- `int_add` and `int_gt` are integer addition and comparison ("greater than"), respectively.

- `guard_true` checks that a boolean is true.

The method calls in the trace are always preceded by a `guard_class` operation, to check that the class of the receiver is the same as the one that was observed during tracing.[5] These guards make the trace specific to the situation where `y` is really a `BoxedInteger`. When the trace is turned into machine code and afterwards executed with `BoxedFloat`, the first `guard_class` instruction will fail and execution will continue using the interpreter.

The trace shows the inefficiencies of `f` clearly, if one looks at the number of `new`, `set/get` and `guard_class` operations. The number of `guard_class` operation is particularly problematic, not only because of the time it takes to run them. All guards also have additional information attached that makes it possible to return to the interpreter, should the guard fail. This means that too many guard operations also consume a lot of memory.

In the rest of the paper we will see how this trace can be optimized using partial evaluation.

3. Object Lifetimes in a Tracing JIT

To understand the problems that this paper is trying to solve in more detail, we first need to understand various cases of object lifetimes that can occur in a tracing JIT compiler.

---

[5] `guard_class` XXX lstinline too large in footnotes performs a precise class check, not checking for subclasses.

[6] We also started to work on optimizing objects in category 3, which will be the subject of a later paper.

```
set(p₅, intval, i₄)                              12
p₆ = new(BoxedInteger)                           15
set(p₆, intval, -100)                            17
```

When the optimizer encounters a `new`, it removes it optimistically, and assumes that the object is in category 1. If later the optimizer finds that the object escapes, it will be allocated at that point. The optimizer needs to keep track of the state of the object that the operation would have created. This is done with the help of a *static object*[7]. The static object describes the shape of the object that would have been allocated, i.e., the type of the object and where the values that would be stored in the fields of the allocated object come from.

In the snippet above, the two `new` operations are removed and two static objects are constructed. The `set` operations manipulate static objects, therefore they can be removed as well; their effect is remembered in the static objects.

After the operations the static object associated with p_5 would store the knowledge that it is a `BoxedInteger` whose `intval` field contains i_4; the one associated with p_6 would store that it is a `BoxedInteger` whose `intval` field contains the constant -100.

The subsequent operations (line 20–26) in Figure 2, which use p_5 and p_6, can then be optimized using that knowledge:

```
guard_class(p₅, BoxedInteger)                    20
i₇ = get(p₅, intval)                             22
guard_class(p₆, BoxedInteger)                    23
i₈ = get(p₆, intval)                             25
i₉ = int_add(i₇, i₈)                             26
```

The `guard_class` operations can be removed, since their arguments are static objects with the matching type `BoxedInteger`. The `get` operations can be removed as well, because each of them reads a field out of a static object. The results of the get operation are replaced with what the static object stores in these fields: all the occurences of i_7 and i_8 in the trace are just replaced by i_4 and -100. The only operation copied into the optimized trace is the addition:

```
i₉ = int_add(i₄, -100)                           26
```

The rest of the trace from Figure 2 is optimized in a similar vein. The operations in lines 27–35 produce two more static objects and are removed. Those in line 36–39 are just copied into the output trace because they manipulate objects that are allocated before the trace. Lines 40–42 are removed because they operate on a static object. Line 43 is copied into the output trace. Lines 44–46 produce a new static object and are removed, lines 48–51 manipulate that static object and are removed as well. Lines 52–54 are copied into the output trace.

The last operation (line 55) is an interesting case. It is the `jump` operation that passes control back to the beginning of the trace. The two arguments to this operation at this point are static objects. However, because they are passed into the next iteration of the loop they live longer than the trace and therefore cannot remain static. They need to be turned into dynamic (runtime) objects before the actual `jump` operation. This process of turning a static object into a dynamic one is called *lifting*.

Lifting a static object puts `new` and `set` operations into the output trace. Those operations produce an object at runtime that has the shape described by the static object. This process is a bit delicate, because the static objects could form an arbitrary graph structure. In our example it is simple, though:

```
p₁₅ = new(BoxedInteger)                          44
```

---

[7] Here "static" is meant in the sense of partial evaluation, i.e., known at partial evaluation time, not in the sense of "static allocation" or "static method".

```
# arguments to the trace: p₀, p₁                 1
guard_class(p₁, BoxedInteger)                    3
i₂ = get(p₁, intval)                             5
guard_class(p₀, BoxedInteger)                    6
i₃ = get(p₀, intval)                             8
i₄ = int_add(i₂, i₃)                             9
i₉ = int_add(i₄, -100)                           26

guard_class(p₀, BoxedInteger)                    37
i₁₂ = get(p₀, intval)                            39
i₁₄ = int_add(i₁₂, -1)                           43

i₁₇ = int_gt(i₁₄, 0)                             52
guard_true(i₁₇)                                  54

p₁₅ = new(BoxedInteger)                          44
set(p₁₅, intval, i₁₄)                            46
p₁₀ = new(BoxedInteger)                          27
set(p₁₀, intval, i₉)                             29

jump(p₁₅, p₁₀)                                   55
```

Figure 4. Resulting Trace After Allocation Removal

```
set(p₁₅, intval, i₁₄)                            46
p₁₀ = new(BoxedInteger)                          27
set(p₁₀, intval, i₉)                             29
jump(p₁₅, p₁₀)                                   55
```

Observe how the operations for creating these two instances have been moved to a later point in the trace. This is worthwhile even though the objects have to be allocated in the end because some `get` operations and `guard_class` operations on the lifted static objects could be removed.

More generally, lifting needs to occur if a static object is used in any operation apart from `get`, `set`, and `guard`. It also needs to occur if `set` is used to store a static object into a non-static one.

The final optimized trace of the example can be seen in Figure 4. The optimized trace contains only two allocations, instead of the original five, and only three `guard_class` operations, compared to the original seven.

5. Formal Description of the Algorithm

In this section we want to give a formal description of the semantics of the traces and of the optimizer and liken the optimization to partial evaluation. We focus on the operations for manipulating heap allocated objects, as those are the only ones that are actually optimized. We also consider only objects with two fields L and R in this section, generalizing to arbitrary many fields is straightforward.

Traces are lists of operations. The operations considered here are `new`, `get`, `set` and `guard_class`. The values of all variables are locations (i.e., pointers). Locations are mapped to objects, which are represented by triples (T, l_1, l_2) of a type T, and two locations that represent the fields of the object. When a new object is created, the fields are initialized to null, but we require that they are initialized to a real location before being read, otherwise the trace is malformed (this condition is guaranteed by how the traces are generated in PyPy).

We use some abbreviations when dealing with object triples. To read the type of an object, $\text{type}((T, l_1, l_2)) = T$ is used. Reading a field F from an object is written $(T, l_1, l_2)_F$ which either is l_1 if $F = L$ or l_2 if $F = R$. To set field F to a new location l, we use the notation $(T, l_1, l_2)!_F l$, which yields a new triple (T, l, l_2) if $F = L$ or a new triple (T, l_1, l) if $F = R$.

$$\text{new} \quad \frac{l \text{ fresh}}{v = \texttt{new}(T), E, H \xRightarrow{\text{run}} E\,[v \mapsto l]\,, H\,[l \mapsto (T, \text{null}, \text{null})]} \qquad\qquad \text{guard} \quad \frac{\text{type}(H(E(v)) = T}{\texttt{guard\_class}(v, T), E, H \xRightarrow{\text{run}} E, H}$$

$$\text{get} \quad \frac{}{u = \texttt{get}(v, F), E, H \xRightarrow{\text{run}} E\,[u \mapsto H\,(E\,(v))_F]\,, H} \qquad\qquad \frac{\text{type}(H(E(v)) \neq T}{\texttt{guard\_class}(v, T), E, H \xRightarrow{\text{run}} \bot, \bot}$$

$$\text{set} \quad \frac{}{\texttt{set}\,(v, F, u)\,, E, H \xRightarrow{\text{run}} E, H\,[E\,(v) \mapsto (H\,(E\,(v))!_F E(u))]}$$

Object Domains:

$u, v \in V$	variables in trace
$T \in \mathfrak{T}$	runtime types
$F \in \{L, R\}$	fields of objects
$l \in L$	locations on heap

Semantic Values:

$E \in V \rightharpoonup L$		Environment
$H \in L \rightharpoonup \mathfrak{T} \times (L \cup \{\text{null}\}) \times (L \cup \{\text{null}\})$	Heap

Figure 5. The Operational Semantics of Simplified Traces

Figure 5 shows the operational semantics for traces. The interpreter formalized there executes one operation at a time. Its state is represented by an environment E and a heap H, which may be changed by the execution of an operation. The environment is a partial function from variables to locations and the heap is a partial function from locations to objects. Note that a variable can never be null in the environment, otherwise the trace would have been malformed. The environment could not directly map variables to objects, because several variables can point to the *same* object, because of aliasing.

We use the following notation for updating partial functions: $E[v \mapsto l]$ denotes the environment which is just like E, but maps v to l.

The new operation creates a new object $(T, \text{null}, \text{null})$ on the heap under a fresh location l and adds the result variable to the environment, mapping it to the new location l.

The get operation reads a field F out of an object, and adds the result variable to the environment, mapping it to the read location. The heap is unchanged.

The set operation changes field F of an object stored at the location that variable v maps to. The new value of the field is the location in variable u. The environment is unchanged.

The guard_class operation is used to check whether the object stored at the location that variable v maps to is of type T. If that is the case, then execution continues without changing heap and environment. Otherwise, execution is stopped.

5.1 Optimizing Traces

To optimize the simple traces of the last section, we use online partial evaluation. The partial evaluator optimizes one operation of a trace at a time. Every operation in the unoptimized trace is replaced by a list of operations in the optimized trace. This list is empty if the operation can be optimized away. The optimization rules can be seen in Figure 6. Lists are written using angular brackets $< ... >$, list concatenation is expressed using two colons: $l_1 :: l_2$.

The state of the optimizer is stored in an environment E and a *static heap* S. Each step of the optimizer takes an operation, an environment and a static heap and produces a list of operations, a new environment and a new static heap.

The environment is a partial function from variables in the unoptimized trace V to variables in the optimized trace V^* (which are themselves written with a * for clarity). The reason for introducing new variables in the optimized trace is that several variables that appear in the unoptimized trace can turn into the same variables in the optimized trace. The environment of the optimizer serves a function similar to that of the environment in the semantics: to express sharing.

The static heap is a partial function from V^* into the set of static objects, which are triples of a type and two elements of V^*. The object referenced by a variable v^* is static, if v^* is in the domain of the static heap S. The object $S(v^*)$ describes what is statically known about the object, i.e., its type and its fields. The fields of objects in the static heap are also elements of V^* (or null, for short periods of time).

When the optimizer sees a new operation, it optimistically removes it and assumes that the resulting object can stay static. The optimization for all further operations is split into two cases. One case is for when the involved variables are in the static heap, which means that the operation can be performed at optimization time and can be removed from the trace. These rules mirror the execution semantics closely. The other case is for when not enough is known about the variables, and the operation has to be residualized.

If the argument v of a get operation is mapped to something in the static heap, the get can be performed at optimization time. Otherwise, the get operation needs to be residualized.

If the first argument v to a set operation is mapped to something in the static heap, then the set can be performed at optimization time (which updates the static heap). Otherwise the set operation needs to be residualized. This needs to be done carefully, because the new value for the field, from the variable u, could itself be static, in which case it needs to be lifted first.

If a guard_class is performed on a variable that is in the static heap, the type check can be performed at optimization time, which means the operation can be removed if the types match. If the type check fails statically or if the object is not in the static heap, the guard_class is residualized. This also needs to lift the variable on which the guard_class is performed.

Lifting takes a variable and turns it into a dynamic variable. If the variable is already dynamic, nothing needs to be done. If it is in the static heap, operations are emitted that construct an object with the shape described there, and the variable is removed from the static heap.

new

$$\frac{v^* \text{ fresh}}{v = \texttt{new}(T), E, S \xRightarrow{\text{opt}} \langle\,\rangle, E\,[v \mapsto v^*], S\,[v^* \mapsto (T, \text{null}, \text{null})]}$$

get

$$\frac{E(v) \in \text{dom}(S)}{u = \texttt{get}(v, F), E, S \xRightarrow{\text{opt}} \langle\,\rangle, E\,[u \mapsto S(E(v))_F], S}$$

$$\frac{E(v) \notin \text{dom}(S), \ u^* \text{ fresh}}{u = \texttt{get}(v, F), E, S \xRightarrow{\text{opt}} \langle u^* = \texttt{get}(E(v), F)\rangle, E\,[u \mapsto u^*], S}$$

set

$$\frac{E(v) \in \text{dom}(S)}{\texttt{set}\,(v, F, u)\,, E, S \xRightarrow{\text{opt}} \langle\,\rangle, E, S\,[E\,(v) \mapsto (S(E(v))!_F\,E(u))]}$$

$$\frac{E(v) \notin \text{dom}\,(S)\,, (E(v), S) \xRightarrow{\text{lift}} (\text{ops}, S')}{\texttt{set}\,(v, F, u)\,, E, S \xRightarrow{\text{opt}} \text{ops} :: \langle\texttt{set}\,(E(v), F, E(u))\rangle, E, S'}$$

guard

$$\frac{E(v) \in \text{dom}(S), \ \text{type}(S(E(v))) = T}{\texttt{guard\_class}(v, T), E, S \xRightarrow{\text{opt}} \langle\,\rangle, E, S}$$

$$\frac{E(v) \notin \text{dom}(S) \vee \text{type}(S(E(v))) \neq T, \ (E(v), S) \xRightarrow{\text{lift}} (\text{ops}, S')}{\texttt{guard\_class}(v, T), E, S \xRightarrow{\text{opt}} \text{ops} :: \langle\texttt{guard\_class}(E\,(v)\,, T)\rangle, E, S'}$$

lifting

$$\frac{v^* \notin \text{dom}(S)}{v^*, S \xRightarrow{\text{lift}} \langle\,\rangle, S}$$

$$\frac{v^* \in \text{dom}(S), \ (v^*, S) \xRightarrow{\text{liftfields}} (\text{ops}, S')}{v^*, S \xRightarrow{\text{lift}} \langle v^* = \texttt{new}\,(\text{type}\,(S\,(v^*)))\rangle :: ops, S'}$$

$$\frac{\left(S\,(v^*)_L\,, S \setminus \{v^* \mapsto S\,(v^*)\}\right) \xRightarrow{\text{lift}} (\text{ops}_L, S')\,, \left(S\,(v^*)_R\,, S'\right) \xRightarrow{\text{lift}} (\text{ops}_R, S'')}{v^*, S \xRightarrow{\text{liftfields}} \text{ops}_L :: ops_R :: \left\langle\texttt{set}\,(v^*, L, S\,(v^*)_L)\,, \texttt{set}\,(v^*, R, S\,(v^*)_R)\right\rangle, S''}$$

Object Domains:

$u, v \in V$	variables in trace
$u^*, v^* \in V^*$	variables in optimized trace
$T \in \mathfrak{T}$	runtime types
$F \in \{L, R\}$	fields of objects

Semantic Values:

$E \in V \rightharpoonup V^*$	Environment
$S \in V^* \rightharpoonup \mathfrak{T} \times (V^* \cup \{\text{null}\}) \times (V^* \cup \{\text{null}\})$	Static Heap

Figure 6. Optimization Rules

Lifting a static object needs to recursively lift its fields. Some care needs to be taken when lifting a static object, because the structures described by the static heap can be cyclic. To make sure that the same static object is not lifted twice, the `liftfield` operation removes it from the static heap *before* recursively lifting its fields.

As an example for lifting, consider the static heap

$$\{v^* \mapsto (T_1, w^*, v^*), w^* \mapsto (T_2, u^*, u^*)\}$$

which contains two static objects. If v^* needs to be lifted, the following residual operations are produced:

```
v* = new(T₁)
w* = new(T₂)
set(w*, L, u*)
set(w*, R, u*)
set(v*, L, w*)
set(v*, R, v*)
```

After the lifting the static heap is the empty set, because both static objects were lifted. If we had lifted w^* instead of v^*, then the following operations would have been produced:

```
w* = new(T₂)
set(w*, L, u*)
set(w*, R, u*)
```

In this case, the static heap afterwards would be:

$$\{v^* \mapsto (T_1, w^*, v^*)\}$$

5.2 Analysis of the Algorithm

While we do not offer a formal proof of it, it can argue informally that the algorithm presented above is sound: it works by delaying (and often completely removing) some operations. The algorithm runs in a single pass over the list of operations. We can check that although recursively lifting a static object is not a constant-time operation, the algorithm only takes a total time linear in the length of the trace. The algorithm itself is not particularly complex; our focus is rather that *in the context of tracing JITs* it is possible to find a simple enough algorithm that performs well.

Note in particular that objects in category 1 (i.e., those that do not escape) are completely removed; moreover, objects in category 2 (i.e., escaping) are still partially optimized: all the operations in between the creation of the object and the point where it escapes that involve the object are removed. Objects in category 3 and 4 are also partially optimized, their allocation is delayed till the end of the trace.

The optimization is particularly effective for chains of operations. For example, it is typical for an interpreter to generate sequences of writes-followed-by-reads, where one interpreted opcode writes to some object's field and the next interpreted opcode reads it back, possibly dispatching on the type of the object created just before. A typical example would be a chain of arithmetic operations.

6. Implementation and Evaluation

The allocation removal techniques described in this paper were implemented in the optimizer of PyPy's tracing JIT. The optimization is independent of which interpreter a JIT is generated for. There are some practical issues beyond the techniques described in this paper. The actual implementation needs to deal with more operations than described in Section 5, e.g., to also support static arrays in addition to static objects. The implementation of this optimization is about 400 lines of RPython code.

A further complication is that most interpreters written with PyPy use heap-allocated frame objects to store local variables.

Those severely hinder the effectiveness of allocation removal, because every time an object is stored into a local variable, it is stored into the frame-object, which makes it escape. We implemented a technique to treat such frames objects in a special way to solve this problem. This is a common approach in VM implementations [13, 19]; the novelty of our approach is that we generalized it enough to be usable for different interpreters.

To evaluate our allocation removal algorithm, we look at the effectiveness when used in the generated tracing JIT of PyPy's Python interpreter. This interpreter is a full implementation of Python 2.5 language semantics and is about 30'000 lines of RPython code.

The benchmarks we used are small-to-medium Python programs, some synthetic benchmarks, some real applications.[8]

Some of them are from the Computer Language Benchmark Game[9]: **fannkuch**, **nbody**, **meteor-contest**, **spectral-norm**.

Furthermore there are the following benchmarks:

- **crypto_pyaes**: An AES implementation.

- **django**: The templating engine of the Django web framework[10].

- **go**: A Monte-Carlo Go AI[11].

- **html5lib**: An HTML5 parser.

- **pyflate-fast**: A BZ2 decoder.

- **raytrace-simple**: A ray tracer.

- **richards**: The Richards benchmark.

- **spambayes**: A Bayesian spam filter[12].

- **telco**: A Python version of the Telco decimal benchmark[13], using a pure Python decimal floating point implementation.

- **twisted_names**: A DNS server benchmark using the Twisted networking framework[14].

We evaluate the allocation removal algorithm along two lines: first we want to know how many allocations could be optimized away. On the other hand, we want to know how much the run times of the benchmarks is improved.

The benchmarks were run on an otherwise idle Intel Core2 Duo P8400 processor with 2.26 GHz and 3072 KB of cache on a machine with 3GB RAM running Linux 2.6.35. We compared the performance of various Python implementations on the benchmarks. As a baseline, we used the standard Python implementation in C, CPython 2.6.6[15], which uses a bytecode-based interpreter. Furthermore we compared against Psyco[21] 1.6, a (hand-written) extension module to CPython which is a just-in-time compiler that produces machine code at run-time. It is not based on traces. Finally, we used two versions of PyPy's Python interpreter (revision 77823 of SVN trunk[16]): one including the JIT but not optimizing

---

[8] All the source code of the benchmarks can be found at `http://codespeak.net/svn/pypy/benchmarks/`. There is also a website that monitors PyPy's performance nightly at `http://speed.pypy.org/`.

[9] `http://shootout.alioth.debian.org/`

[10] `http://www.djangoproject.com/`

[11] `http://shed-skin.blogspot.com/2009/07/disco-elegant-python-go-player.html`

[12] `http://spambayes.sourceforge.net/`

[13] `http://speleotrove.com/decimal/telco.html`

[14] `http://twistedmatrix.com/`

[15] `http://python.org`

[16] `http://codespeak.net/svn/pypy/trunk`

the traces, and one using the allocation removal optimizations (as well as some minor other optimizations, such as constant folding).

As the first step, we counted the occurring operations in all generated traces before and after the optimization phase for all benchmarks. The resulting numbers can be seen in Figure 7. The optimization removes between 4% and 90% and of allocation operations in the traces of the benchmarks. All benchmarks taken together, the optimization removes 70% percent of allocation operations. The numbers look similar for reading and writing of attributes. There are even more `guard` operations that are removed, however there is an additional optimization that removes guards, so not all the removed guards are an effect of the optimization described here (for technical reasons, it would be very hard to separate the two effects).

In addition to the count of operations we also performed time measurements. All benchmarks were run 50 times in the same process, to give the JIT time to produce machine code. The arithmetic mean of the times of the last 30 runs were used as the result. The errors were computed using a confidence interval with a 95% confidence level [16]. The results are reported in Figure 8. For each implementation the table also reports the speedup that PyPy with optimization achieves over it.

With the optimization turned on, PyPy's Python interpreter outperforms CPython in all benchmarks except spambayes (which heavily relies on regular expression performance and thus is not helped much by our Python JIT) and meteor-contest. All benchmarks are improved by the allocation removal optimization, by at least 20% and by as much as a factor of 6.95.

Psyco is able to outperform PyPy's JIT in five out of 14 benchmarks. We hope to overtake Psyco (which is no longer being actively developped) by adding some further optimizations.

7. Related Work

There exists a large number of works on escape analysis, which is a program analysis that tries to find an upper bound for the lifetime of objects allocated at specific program points [4, 11, 17, 20]. This information can then be used to decide that certain objects can be allocated on the stack, because their lifetime does not exceed that of the stack frame it is allocated in. The difference to our work is that escape analysis is split into an analysis and an optimization phase. The analysis can be a lot more complex than our simple one-pass optimization. Also, stack-allocation reduces garbage-collection pressure but does not optimize away the actual accesses to the stack-allocated object. In our case, an object is not needed at all any more.

Chang *et al.* describe a tracing JIT for JavaScript running on top of a JVM [10]. They mention in passing an approach to allocation removal that moves the allocation of an object of type 1 out of the loop to only allocate it once, instead of every iteration. No details are given for this optimization. The fact that the object is still allocated and needs to be written to means that only the allocations are optimized away, but not the reads out of and writes into the object.

SPUR, a tracing JIT for C# seems to be able to remove allocations in a similar way to the approach described here, as hinted at in the technical report [3]. However, no details for the approach and its implementation are given.

Psyco [21] is a (non-tracing) JIT for Python that implements a more ad-hoc version of the allocation removal described here. Our static objects could be related to what are called *virtual* objects in Psyco. Historically, PyPy's JIT can be seen as some successor of Psyco for a general context (one of the authors of this paper is the author of Psyco).

The original SELF JIT compiler [9] used an algorithm for forward-propagating the types of variables as part of its optimiza-

tions. This makes it possible to remove all type checks on a variable but the first one. The optimization does not deal with removing the full object, if it is short-lived, but the type check removals are similar to what our optimization achieves.

Partially known data structures are built directly into Prolog (via unbound logic variables) and thus the treatment of partially static data structures was part of partial evaluation of Prolog programs from the early stages [18]. One effect of unfolding in Prolog is that terms that are constructed and immediately matched again, completely disappear in the residual program. This is similar to what our optimization does for an imperative language. In functional programming this idea was introduced as constructor specialisation by Mogensen [**?**].

8. Conclusion and Future Work

In this paper, we used an approach based on online partial evaluation to optimize away allocations and type guards in the traces of a tracing JIT. In this context a simple approach based on partial evaluation gives good results. This is due to the fact that the tracing JIT itself is responsible for all control issues, which are usually the hardest part of partial evaluation: the tracing JIT selects the parts of the program that are worthwhile to optimize, and extracts linear paths through them, inlining functions as necessary. What is left to optimize are only those linear paths.

We expect a similar result for other optimizations that usually require a complex analysis phase and are thus normally too slow to use at runtime. A tracing JIT selects interesting linear paths by itself; therefore, a naive version of many optimizations on such paths should give mostly the same results. For example, we experimented with (and plan to write about) store-load propagation with a very simple alias analysis.

Acknowledgements

References

[1] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *Proceedings of the 2007 symposium on Dynamic languages*, pages 53–64, Montreal, Quebec, Canada, 2007. ACM.

[2] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[3] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. SPUR: a trace-based JIT compiler for CIL. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 708–725, Reno/Tahoe, Nevada, USA, 2010. ACM.

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

[5] C. F. Bolz, A. Cuni, M. Fijałkowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25, Genova, Italy, 2009. ACM.

[6] C. F. Bolz, A. Kuhn, A. Lienhard, N. Matsakis, O. Nierstrasz, L. Renggli, A. Rigo, and T. Verwaest. Back to the future in one week — implementing a Smalltalk VM in PyPy. In *Self-Sustaining Systems*, pages 123–139. 2008.

[7] C. F. Bolz, M. Leuschel, and D. Schneider. Towards a jitting VM for Prolog execution. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 99–108, Hagenberg, Austria, 2010. ACM.

	num loops	new	removed	get/set	removed	guard	removed	all ops	removed
crypto_pyaes	78	3088	50%	57148	25%	9055	95%	137189	80%
django	51	673	54%	19318	18%	3876	93%	55682	85%
fannkuch	43	171	49%	886	63%	1159	81%	4935	45%
go	517	12234	76%	200842	21%	53138	90%	568542	84%
html5lib	498	14432	68%	503390	11%	71592	94%	1405780	91%
meteor-contest	59	277	36%	4402	31%	1078	83%	12862	68%
nbody	13	96	38%	443	69%	449	78%	2107	38%
pyflate-fast	162	2278	55%	39126	20%	8194	92%	112857	80%
raytrace-simple	120	3118	59%	91982	15%	13572	95%	247436	89%
richards	87	844	4%	49875	22%	4130	91%	133898	83%
spambayes	314	5608	79%	117002	11%	25313	94%	324125	90%
spectral-norm	38	360	64%	5553	20%	1122	92%	11878	77%
telco	46	1257	90%	37470	3%	6644	99%	98590	97%
twisted-names	214	5273	84%	100010	10%	23247	96%	279667	92%
total	2240	49709	70%	1227447	14%	222569	93%	3395548	89%

Figure 7. Number of Operations and Percentage Removed By Optimization

	CPython [ms]	×	Psyco [ms]	×	PyPy w/o optim. [ms]	×	PyPy w/ optim. [ms]	×
crypto_pyaes	2757.80 ± 0.98	10.33	67.90 ± 0.47	0.25	1652.00 ± 4.00	6.19	266.86 ± 5.94	1.00
django	993.19 ± 0.50	3.83	913.51 ± 4.22	3.52	694.73 ± 2.86	2.68	259.53 ± 1.79	1.00
fannkuch	1987.22 ± 2.02	4.26	944.44 ± 0.61	2.02	566.99 ± 1.06	1.21	466.87 ± 1.85	1.00
go	947.21 ± 1.58	3.00	445.96 ± 0.68	1.41	2197.71 ± 25.21	6.95	316.15 ± 9.33	1.00
html5lib	13987.12 ± 19.51	1.39	17398.25 ± 36.50	1.72	27194.45 ± 46.62	2.69	10092.19 ± 23.50	1.00
meteor-contest	346.98 ± 0.35	0.88	215.66 ± 0.23	0.55	433.04 ± 1.45	1.10	392.85 ± 0.87	1.00
nbody_modified	637.90 ± 1.82	6.14	256.78 ± 0.18	2.47	135.55 ± 0.33	1.30	103.93 ± 0.25	1.00
pyflate-fast	3169.35 ± 1.89	1.74	1278.16 ± 3.13	0.70	3285.89 ± 8.51	1.80	1822.36 ± 11.52	1.00
raytrace-simple	2744.60 ± 51.72	4.24	1072.66 ± 1.08	1.66	2778.27 ± 15.13	4.29	647.24 ± 5.44	1.00
richards	354.06 ± 1.00	4.01	63.48 ± 0.15	0.72	383.93 ± 3.28	4.35	88.32 ± 0.91	1.00
spambayes	299.16 ± 0.35	0.75	338.68 ± 3.14	0.85	580.90 ± 24.68	1.46	397.37 ± 10.60	1.00
spectral-norm	478.63 ± 0.80	4.27	139.83 ± 1.54	1.25	353.51 ± 1.39	3.15	112.10 ± 1.17	1.00
telco	1207.67 ± 2.03	2.44	730.00 ± 2.66	1.47	1296.08 ± 4.37	2.62	495.23 ± 2.14	1.00
twisted_names	9.58 ± 0.01	1.34	10.43 ± 0.01	1.46	17.99 ± 0.27	2.52	7.13 ± 0.09	1.00

Figure 8. Benchmark Times in Milliseconds, Together With Factor Over PyPy With Optimizations

[8] C. Bruni and T. Verwaest. PyGirl: generating Whole-System VMs from High-Level prototypes using PyPy. In W. Aalst, J. Mylopoulos, N. M. Sadeh, M. J. Shaw, C. Szyperski, M. Oriol, and B. Meyer, editors, *Objects, Components, Models and Patterns*, volume 33 of *Lecture Notes in Business Information Processing*, pages 328–347. Springer Berlin Heidelberg, 2009. 10.1007/978-3-642-02571-6_19.

[9] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on proto-types. *SIGPLAN Not.*, 24(10):49–70, 1989.

[10] M. Chang, M. Bebenita, A. Yermolovich, A. Gal, and M. Franz. Efficient just-in-time execution of dynamically typed languages via code specialization using precise runtime type inference. Technical Report ICS-TR-07-10, Donald Bren School of Information and Computer Science, University of California, Irvine, 2007.

[11] J. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. *SIGPLAN Not.*, 34(10):1–19, 1999.

[12] A. Cuni. *High performance implementation of Python for CLI/.NET with JIT compiler generation for dynamic languages.* PhD thesis, Dipartimento di Informatica e Scienze dell'Informazione, University of Genova, 2010. Technical Report DISI-TH-2010-05.

[13] A. Gal, B. Eich, M. Shaver, D. Anderson, B. Kaplan, G. Hoare, D. Mandelin, B. Zbarsky, J. Orendorff, M. Bebenita, M. Chang, M. Franz, E. Smith, R. Reitmaier, and M. Haghighat. Trace-based just-in-time type specialization for dynamic languages. In *PLDI*, 2009.

[14] A. Gal and M. Franz. Incremental dynamic code generation with trace trees. Technical Report ICS-TR-06-16, Donald Bren School of Information and Computer Science, University of California, Irvine, Nov. 2006.

[15] A. Gal, C. W. Probst, and M. Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 144–153, Ottawa, Ontario, Canada, 2006. ACM.

[16] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. *SIGPLAN Not.*, 42(10):57–76, 2007.

[17] B. Goldberg and Y. G. Park. Higher order escape analysis: optimizing stack allocation in functional program implementations. In *Proceedings of the third European symposium on programming on ESOP '90*, pages 152–160, Copenhagen, Denmark, 1990. Springer-Verlag New York, Inc.

[18] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *J. Log. Program.*, 11(3-4):217–242, 1991.

[19] E. Miranda. Context management in VisualWorks 5i. Technical report, ParcPlace Division, CINCOM, Inc., 1999.

[20] Y. G. Park and B. Goldberg. Escape analysis on lists. *SIGPLAN Not.*, 27(7):116–127, 1992.

[21] A. Rigo. Representation-based just-in-time specialization and the Psyco prototype for Python. In *Proceedings of the 2004 ACM SIG-PLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 15–26, Verona, Italy, 2004. ACM.

[22] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 944–953, Portland, Oregon, USA, 2006. ACM.