Allocation Removal by Partial Evaluation in a Tracing JIT

Carl Friedrich Bolz Antonio Cuni Maciej Fijałkowski Michael Leuschel Samuele Pedroni
Armin Rigo

Heinrich-Heine-Universität Düsseldorf, STUPS Group, Germany XXX

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 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 goal of a just-in-time (JIT) compiler for a dynamic language is obviously to improve the speed of the language over an implementation of the language that uses interpretation. The first goal of a JIT is thus 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 of 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 a new box, in addition to the actual computation they do. Because the boxes are allocated on the heap, producing a lot 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 doing a generic operation on them. An example would be the addition of two objects: The addition needs to check what the concrete objects that should be added are, and choose the implementation that is fitting for them. Type dispatching is a very common operation in a dynamic language because no types are known at compile time, so 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, which are then turned into machine code. One reason for the popularity of tracing JITs is their relative simplicity. They can often be added to an interpreter and a lot of the infrastructure of an interpreter can be reused. They give some important optimizations like inlining and constant-folding for free. A tracing JIT always produces linear pieces of code, which simplifies many optimizations that are usually hard in a compiler, such as register allocation.

The usage 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 make some objects that are allocated in the trace *static*[2] 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.

In Section 6 we describe some supporting techniques that are not central to the approach, but are needed to improve the results. The introduced techniques are evaluated in Section 7 using PyPy's Python interpreter as a case study.

The contributions of this paper are:

1. An efficient and effective algorithm for removing object allocations in a tracing JIT.
2. A characterization of this algorithm as partial evaluation.
3. A rigorous evaluation of this algorithm.

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

[2] These objects are called *virtual* in Psyco [21].

PEPM'11, XXX

2. Background

2.1 PyPy

The work described in this paper was done in the context of the PyPy project [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 the C level. Because the interpreter is written at a relatively high level, the language implementation is kept free of low-level details, such as object layout, garbage collection or memory model. Those aspects of the final VM are woven into the generated code during the translation to C.

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 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. Mostly-automatically generating a JIT compiler has many advantages over writing one manually, which is an error-prone and tedious process. By construction, the generated JIT has the same semantics as the interpreter. Many optimizations can benefit all languages implemented as an interpreter in RPython.

Moreover, thanks to the internal design of the JIT generator, it is very easy to add new *backends* for producing the actual machine code, in addition to the original backend for the Intel *x86* architecture. Examples of additional JIT backends are the one for Intel *x86-64* and an experimental one for the CLI .NET Virtual Machine [11]. PyPy's JIT generator generates a *tracing JIT compiler*, a concept which we now explain in more details.

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 [14] and afterwards used in several implementations of dynamic languages, such as JavaScript [12], Lua[3] 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 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 thus 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 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 is then the basis of the generated code. The trace is first optimized, and then turned into machine code. Both optimizations and machine code generation is 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 it. To make sure that the trace is maintaining 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 true, because if it isn't, the rest of the trace is not valid.

When generating machine code, every guard is be 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. The existing assembler is patched to jump to the new trace when the guard fails [13]. 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 that a dynamic language implementation has. All the numbers are instances of either `BoxedInteger` or `BoxedFloat`, thus they consume space on the heap. Performing many arithmetic operations produces lots of garbage quickly, thus 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.

To understand the problems more directly, let us consider the simple interpreter function `f` that uses the object model (see the bottom of Figure 1).

XXX this is not an RPython interpreter; put a reference to the previous paper to show how we deal with an interpreted piece of code and remove the interpretation overhead, turning it into basically something equivalent to the example here, which is the start of the present paper.

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 do not survive for very long. The actual

[3] `http://luajit.org/`

```
class Base(object):
      ...

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

computation that is performed by `f` is simply a sequence of float or integer additions.

If the function is executed using the tracing JIT, with `y` being a `BoxedInteger`, the produced trace looks like Figure 2 (lines starting with the hash "#" are comments).

XXX in which language is the trace written in ? still RPython ?

The operations in the trace are shown indented to correspond 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 to exactly once. The arguments p_0 and p_1 of the loop correspond to the live variables `y` and `res` in the original function.

XXX explain set and get + int_add briefly

The trace shows the inefficiencies of `f` clearly, if one looks at the number of **`new`** (corresponding to object creation), **`set/get`** (corresponding to attribute reads/writes) and **`guard_class`** operations

```
# arguments to the trace: p0, p1
# inside f: res.add(y)
guard_class(p1, BoxedInteger)
    # inside BoxedInteger.add
    i2 = get(p1, intval)
    guard_class(p0, BoxedInteger)
        # inside BoxedInteger.add__int
        i3 = get(p0, intval)
        i4 = int_add(i2, i3)
        p5 = new(BoxedInteger)
            # inside BoxedInteger.__init__
            set(p5, intval, i4)

# inside f: BoxedInteger(-100)
p6 = new(BoxedInteger)
    # inside BoxedInteger.__init__
    set(p6, intval, -100)

# inside f: .add(BoxedInteger(-100))
guard_class(p5, BoxedInteger)
    # inside BoxedInteger.add
    i7 = get(p5, intval)
    guard_class(p6, BoxedInteger)
        # inside BoxedInteger.add__int
        i8 = get(p6, intval)
        i9 = int_add(i7, i8)
        p10 = new(BoxedInteger)
            # inside BoxedInteger.__init__
            set(p10, intval, i9)

# inside f: BoxedInteger(-1)
p11 = new(BoxedInteger)
    # inside BoxedInteger.__init__
    set(p11, intval, -1)

# inside f: y.add(BoxedInteger(-1))
guard_class(p0, BoxedInteger)
    # inside BoxedInteger.add
    i12 = get(p0, intval)
    guard_class(p11, BoxedInteger)
        # inside BoxedInteger.add__int
        i13 = get(p11, intval)
        i14 = int_add(i12, i13)
        p15 = new(BoxedInteger)
            # inside BoxedInteger.__init__
            set(p15, intval, i14)

# inside f: y.is_positive()
guard_class(p15, BoxedInteger)
    # inside BoxedInteger.is_positive
    i16 = get(p15, intval)
    i17 = int_gt(i16, 0)
# inside f
guard_true(i17)
jump(p15, p10)
```

Figure 2. Unoptimized Trace for the Simple Object Model

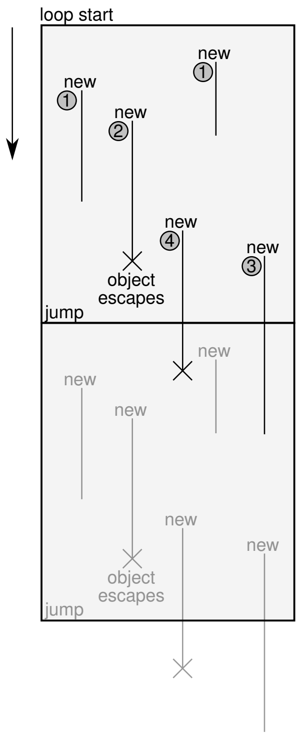

Figure 3. Object Lifetimes in a Trace

(corresponding to method calls). In the rest of the paper we will see how this trace can be optimized using partial evaluation.

Note how the functions that are called by `f` are automatically inlined into the trace. The method calls 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.[4] These guards make the trace specific to the situation where `y` is really a `BoxedInteger`, it can already be said to be specialized for `BoxedIntegers`. When the trace is turned into machine code and then executed with `BoxedFloats`, the first **`guard_class`** instruction will fail and execution will continue using the interpreter.

3. Object Lifetimes in a Tracing JIT

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

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 a while, 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.

[4] **`guard_class`** performs a precise class check, not checking for subclasses.

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:

- Category 1: Objects that live for a while, and are then just not used any more.
- Category 2: Objects that live for a while and then escape.
- Category 3: Objects that live for a while, survive across the jump to the beginning of the loop, and are then not used any more.
- Category 4: Objects that live for a while, 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.

4. Allocation Removal in Traces

4.1 Static Objects

The main insight to improve the code shown in the last section is that objects in category 1 don't survive very long – they are used only inside the loop and nobody else in the program stores a reference to them. The idea for improving the code is thus to analyze which objects fall in category 1 and may thus not be allocated at all.

This is a process that is usually called *escape analysis*. In this paper we will perform escape analysis by using partial evaluation. The partial evalution is a bit peculiar in that there are not actually any constant arguments to the trace, but it is only used to optimized operations within a trace. XXX mention Prolog.

The partial evaluation works by walking the trace from beginning to end. Whenever a **`new`** operation is seen, the operation is removed and a static object is constructed and associated with the variable that would have stored the result of **`new`**. The static object describes the shape of the original object, e.g., where the values that would be stored in the fields of the allocated object come from, as well as the type of the object. Whenever the optimizer sees a **`set`** that writes into such an object, that shape description is updated and the operation can be removed, which means that the operation was done at partial evaluation time. When the optimizer encounters a **`get`** from such an object, the result is read from the shape description, and the operation is also removed. Equivalently, a **`guard_class`** on a variable that has a shape description can be removed as well, because the shape description stores the type and thus the outcome of the type check the guard does is statically known.

In the example from last section, the following operations would produce two static objects, and be completely removed from the optimized trace:

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

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

The following operations on p_5 and p_6 could then be optimized using that knowledge:

```
guard_class(p5, BoxedInteger)
i7 = get(p5, intval)
# inside BoxedInteger.add
guard_class(p6, BoxedInteger)
# inside BoxedInteger.add__int
i8 = get(p6, intval)
i9 = int_add(i7, i8)
```

The **guard_class** operations can be removed, because the classes of p_5 and p_6 are known to be `BoxedInteger`. The **get** operations can be removed and i_7 and i_8 are just replaced by i_4 and -100. Thus the only remaining operation in the optimized trace would be:

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

The rest of the trace is optimized similarly.

So far we have only described what happens when static objects are used in operations that read and write their fields and in guards. When the static object is used in any other operation, it cannot stay static. For example, when a static object is stored in a globally accessible place, the object needs to actually be allocated, as it might live longer than one iteration of the loop and because the partial evaluator looses track of it. This means that the static object needs to be turned into a dynamic one, i.e., lifted. This makes it necessary to put operations into the residual code that actually allocate the static object at runtime.

This is what happens at the end of the trace in Figure 2, when the `jump` operation is hit. The arguments of the jump are at this point static objects. Before the jump is emitted, they are *lifted*. This means that the optimizer produces code that allocates a new object of the right type and sets its fields to the field values that the static object has (if the static object points to other static objects, those need to be lifted as well, recursively) This means that instead of the jump, the following operations are emitted:

```
p15 = new(BoxedInteger)
set(p15, intval, i14)
p10 = new(BoxedInteger)
set(p10, intval, i9)
jump(p15, p10)
```

Note how the operations for creating these two instances have been moved down the trace. It looks like for these operations we actually didn't win much, because the objects are still allocated at the end. However, the optimization was still worthwhile even in this case, because some operations that have been performed on the lifted static objects have been removed (some **get** operations and **guard_class** operations).

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, from 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 concentrate on the operations for manipulating dynamically allocated objects, as those are the only ones that are actually optimized. Without loss of generality we also consider only objects with two fields in this section.

Traces are lists of operations. The operations considered here are **new** (to make a new object), **get** (to read a field out of an object), **set** (to write a field into an object) and **guard_class**

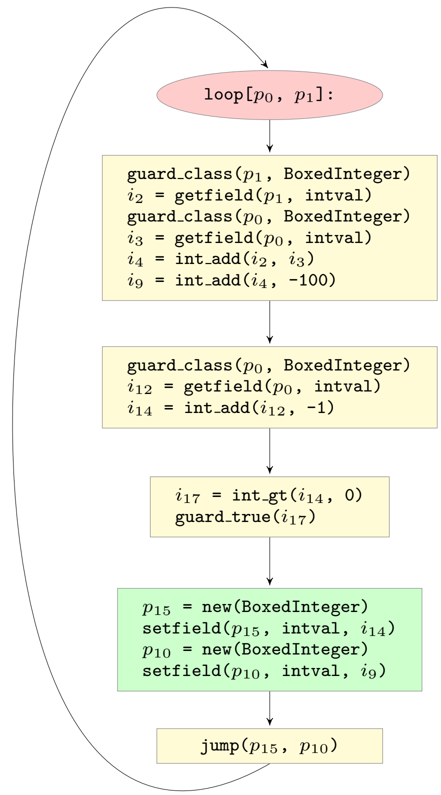

Figure 4. Resulting Trace After Allocation Removal

(to check the type of an object). The values of all variables are locations (i.e. pointers). Locations are mapped to objects, which are represented by triples 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.

We use some abbreviations when dealing with object triples. To read the type of an object, $\mathrm{type}((T, l_1, l_2)) = T$ is used. Reading a field F from an object is written $(T, l_1, l_2)_F$ which either returns 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 $F = R$.

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 are potentially 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 be malformed. The environment could not directly map variables to object, because several variables can contain a pointer to the *same* object. Thus the "indirection" is needed to express sharing.

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 $(T, \mathrm{null}, \mathrm{null})$ 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.

$$\textit{new}\quad \frac{l \text{ fresh}}{v = \texttt{new}(T), E, H \overset{\text{run}}{\Longrightarrow} E\left[v \mapsto l\right], H\left[l \mapsto (T, \text{null}, \text{null})\right]}$$

$$\textit{guard}\quad \frac{\text{type}(H(E(v)) = T}{\texttt{guard\_class}(v, T), E, H \overset{\text{run}}{\Longrightarrow} E, H}$$

$$\textit{get}\quad \frac{}{u = \texttt{get}(v, F), E, H \overset{\text{run}}{\Longrightarrow} E\left[u \mapsto H\left(E\left(v\right)\right)_F\right], H}$$

$$\frac{\text{type}(H(E(v)) \neq T}{\texttt{guard\_class}(v, T), E, H \overset{\text{run}}{\Longrightarrow} \bot, \bot}$$

$$\textit{set}\quad \frac{}{\texttt{set}\left(v, F, u\right), E, H \overset{\text{run}}{\Longrightarrow} E, H\left[E\left(v\right) \mapsto \left(H\left(E\left(v\right)\right)!_F E(u)\right)\right]}$$

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

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 from the last section, we use online partial evaluation. The partial evaluator optimizes one operation of the 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 (which hopefully happens often). The optimization rules can be seen in Figure 6.

The state of the optimizer is stored in an environment E and a *static heap* S. 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. Thus the environment of the optimizer serves a function similar to that of the environment in the semantics: 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^*. A variable v^* is in the domain of the static heap S as long as the optimizer can fully keep track of the object. The image of v^* is what is statically known about the object stored in it, 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 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 performed at optimization time and the static heap updated. 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 that is potentially in the static heap and makes sure that it is turned into a dynamic variable. This means that operations are emitted that construct an object with the shape described in the static heap, and the variable is removed from the static heap.

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.

5.2 Analysis of the Algorithm

While we do not offer a formal proof of it, it should be relatively clear 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 altough 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. Moreover, it gives the "best" possible result within its constrains, e.g., in term of the number of residual operations. The algorithm itself is not particularly complex or innovative; our focus is rather that *in the context of tracing JITs* it is possible to find a simple enough algorithm that still gives the best results.

Note in particular that objects in category 1 (i.e., the ones that do not escape) are completely removed; moreover, objects in category 2 (i.e., escaping) are still partially dealt with: if such an object escapes later than its creation point, all the operations inbetween that involve the object are removed.

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

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

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

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

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

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

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

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

$$\frac{v^* \in \text{dom}(S),\ (v^*, S) \overset{\text{liftfields}}{\Longrightarrow} (\text{ops}, S')}{v^*, S \overset{\text{lift}}{\Longrightarrow} \langle v^* = \texttt{new}\left(T\right)\rangle :: \mathit{ops}, S'}$$

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

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

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. In the case of PyPy's Python interpreter, this optimization can even remove the allocation of all intermediate frames that occur in the interpreter, corresponding to all calls that have been inlined in the trace.

6. Supporting Techniques

6.1 Virtualizables

CFB ►*probably can be cut in case of space problems*◄

One problem to the successful application of the allocation removal techniques described in the previous sections is the presence of frame-introspection features in many dynamic languages. Languages such as Python and Smalltalk allow the programmer to get access to the frames object that the interpreter uses to store local variables. This is a useful feature, as makes the implementation of a debugger possible in Python without needing much support from the VM level. On the other hand, it severely hinders 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.

This problem is solved by making it possible to the interpreter author to add some hints into the source code to declare instances of one class as frame objects. The JIT will then fill these objects only lazily when they are actually accessed (e.g., because a debugger is used). Therefore in the common case, nothing is stored into the frame objects, making the problem of too much escaping go away. This is a common approach in VM implementations [19], the only novelty in our approach lays in its generality, because most other JITs are just specifically written for one particular language.

7. Evaluation

To evaluate the effectiveness of our allocation removal algorithm, we look at the effectiveness when used in the tracing JIT of PyPy's Python interpreter. The benchmarks we used are small-to-medium Python programs, some synthetic benchmarks, some real applications.

Some of them are from the Computer Language Benchmark Game[5]: **fannkuch**, **nbody**, **meteor-contest**, **spectral-norm**.

Furthermore there are the following benchmarks:

- **crypto_pyaes**: AES implementation.
- **django**: The templating engine of the Django web framework[6].
- **go**: A Monte-Carlo Go AI[7].
- **html5lib**: An HTML5 parser.
- **pyflate-fast**: A BZ2 decoder.
- **raytrace-simple**: A ray tracer.
- **richards**: The Richards benchmark.
- **spambayes**: A Bayesian spam filter[8].
- **telco**: A Python version of the Telco decimal benchmark[9], using a pure Python decimal floating point implementation.
- **twisted_names**: A DNS server benchmark using the Twisted networking framework[10].

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.

For the former we counted the occurring operations in all generated traces before and after the optimization phase for all benchmarks. The results can be seen in Figure 7. The optimization removes as many as XXX and as little as XXX percent of allocation operations in the traces of the benchmarks. All benchmarks taken together, the optimization removes XXX percent of allocation operations.

In addition to the count of operations we also performed time measurements. The machine the benchmarks were run on is XXX. We compared the performance of various Python implementations on the benchmarks. As a baseline, we used the standard Python implementation in C, called CPython[11], which uses a bytecode-based interpreter. Furthermore we compared against Psyco [21], an extension 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 three versions of PyPy's Python interpreter: one without a JIT, one including the JIT but not using the allocation removal optimization, and one using the allocation removal optimizations.

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 [15]. The results are reported in Figure 8. With the optimization turned on, PyPy's Python interpreter outperforms CPython in all benchmarks except spambayes (which heavily relies on regular expression performance). All benchmarks are improved by the allocation removal optimization, some as much as XXX. XXX Psyco

XXX runtimes of the algorithm somehow?

8. Related Work

There exists a large number of works on escape analysis, which is an program analysis that tries to find an upper bound for the lifetime of objects allocated at specific program points [4, 10, 16, 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. XXX more papers?

Chang *et al.* describe a tracing JIT for JavaScript running on top of a JVM [9]. 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 and writes out of/into the object.

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

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

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

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

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

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

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

	#loops	new			get			set			guard			rest	
crypto_pyaes	75	1663	446	27%	18842	2127	11%	14552	2020	14%	8952	235	3%	53339	
django	53	453	155	34%	11970	1457	12%	8738	991	11%	4093	275	7%	33988	
fannkuch	40	161	76	47%	574	306	53%	278	139	50%	1121	202	18%	2581	
go	510	5604	1651	29%	112403	12612	11%	88439	11915	13%	51621	3194	6%	300831	3
html5lib	416	5939	1793	30%	241402	15863	7%	178723	13960	8%	58486	3037	5%	682654	4
meteor-contest	56	221	141	64%	2326	606	26%	2047	645	32%	1040	169	16%	7018	
nbody	10	84	48	57%	287	159	55%	121	74	61%	411	90	22%	986	
pyflate-fast	161	1878	723	38%	22815	2592	11%	18415	3690	20%	8291	628	8%	64649	1
raytrace-simple	117	1906	359	19%	52995	2202	4%	39518	1823	5%	13451	447	3%	140723	
richards	79	375	191	51%	27507	2095	8%	21217	2004	9%	3972	332	8%	76751	
spambayes	267	2413	505	21%	53327	3033	6%	40938	2428	6%	19343	718	4%	143575	1
spectral-norm	35	266	88	33%	3004	398	13%	2544	414	16%	1084	78	7%	4803	
telco	42	660	70	11%	20716	424	2%	14444	608	4%	6574	56	1%	53674	
twisted-names	201	2706	317	12%	64438	2795	4%	47960	1968	4%	24091	555	2%	171582	
total	2062	24329	6563	27%	632606	46669	7%	477934	42679	9%	202530	10016	5%	1737154	13

Figure 7. Number of Operations Before and After Optimization

	CPython	JIT no optimizations	JIT allocation removal	JIT full
crypto_pyaes	1958.40 ± 0.32	782.14 ± 3.08	150.86 ± 3.62	125.68 ± 3.02
django	630.91 ± 0.13	365.90 ± 1.57	124.41 ± 0.88	117.36 ± 0.91
fannkuch	1344.15 ± 0.36	378.18 ± 0.36	309.04 ± 1.56	299.12 ± 0.25
go	595.31 ± 0.51	1067.26 ± 12.20	121.76 ± 4.15	125.57 ± 3.90
html5lib	8678.67 ± 23.37	17665.16 ± 2695.45	6739.95 ± 1225.25	6699.24 ± 1295.92
meteor-contest	241.53 ± 0.09	291.11 ± 0.86	274.09 ± 0.37	272.77 ± 0.30
nbody	396.55 ± 0.24	84.77 ± 0.14	69.01 ± 0.08	68.71 ± 0.09
pyflate-fast	1991.33 ± 1.90	1818.70 ± 4.30	1062.03 ± 4.56	1052.02 ± 3.24
raytrace-simple	1598.84 ± 1.07	1126.93 ± 11.42	93.45 ± 4.75	92.17 ± 4.15
richards	216.47 ± 0.20	169.84 ± 1.21	11.81 ± 0.09	11.90 ± 0.29
spambayes	193.30 ± 0.03	331.37 ± 14.93	238.83 ± 6.10	237.23 ± 5.97
spectral-norm	375.91 ± 0.08	221.24 ± 0.07	31.96 ± 0.06	28.79 ± 0.09
telco	789.67 ± 1.14	622.67 ± 1.86	155.33 ± 1.82	154.67 ± 1.82
twisted_names	6.61 ± 0.00	8.85 ± 0.09	4.35 ± 0.05	4.35 ± 0.04

Figure 8. Benchmark Times

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.

partial evaluation:

Prolog: partially static datastructures are already built-in to Prolog and was built-into partial evaluation from the beginning [17]. FP

partially static data structures: kenichi asai's thesis?

xxx: relation to compile-time garbage collection [18]; separation logic; John Hughes: type specialization

9. Conclusion

In this paper, we used a approach based on online partial evaluation to optimize allocations in the traces of a tracing JIT. In this context a simple approach to 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 is 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

The authors would like to thank Stefan Hallerstede and Thomas Stiehl for fruitful discussions during the writing of the paper.

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. Technical Report MSR-TR-2010-27, Microsoft Research, Mar. 2010.

[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. Hagenberg, Austria, 2010. accepted for publication.

[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] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

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

[18] N. Mazur, P. Ross, G. Janssens, and M. Bruynooghe. Practical aspects for a working compile time garbage collection system for mercury. In P. Codognet, editor, *Logic Programming*, volume 2237 of *Lecture Notes in Computer Science*, pages 105–119. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-45635-X_15.

[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 SIGPLAN 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.