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, samuele.pedroni@gmail.com, arigo@tunes.org

Abstract

[1]

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

General Terms Languages, Performance, Experimentation

Keywords XXX

XXX drop the word "allocation removal" somewhere
XXX define "escape analysis"

1. Introduction

XXX need to re-target introduction a bit to fit PEPMs focus

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 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 makes many optimizations that are usually hard in a compiler simpler, 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 an approach how an optimization can be added to a tracing JIT that also 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 most important technique we use to achieve to optimize traces is a straightforward application of partial evaluation. The partial evaluation performs a form of escape analysis [citation needed] 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.

The basic approach of static objects can then be extended to also be used for type-specializing the traces that are produced by the tracing JIT (Section 6). In Section 7 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 8 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).

PEPM'11, XXX

[2] These objects are called *virtual* in Psyco [10].

2. Background

2.1 PyPy

The work described in this paper was done in the context of the PyPy project [11]. PyPy is an environment where dynamic languages can be implemented in a simple yet efficient way. The approach taken when implementing a language with PyPy is to write 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 started 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 [5], a Smalltalk VM [4] and a GameBoy emulator [6].

The feature that makes PyPy more than a compiler with a runtime system is it's support for automated JIT compiler generation [3]. 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 by 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. The JIT that is produced by PyPy's JIT generator is 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 [9] and afterwards used in several implementations of dynamic languages, such as JavaScript [7], Lua [citation needed] 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 records 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 optimized in some ways, 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 [8].

2.3 Running Example

For the purpose of this paper, we are going to use 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. These classes could be part of the implementation of a very simple interpreter 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 a simple function that uses the object model:

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.

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

The loop 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 computation that is performed by `f` is simply a number of float or integer additions.

```
class Base(object):
    def add(self, other):
        """ add self to other """
        raise NotImplementedError("abstract base")
    def add__int(self, intother):
        """ add intother to self,
            where intother is an integer """
        raise NotImplementedError("abstract base")
    def add__float(self, floatother):
        """ add floatother to self,
            where floatother is a float """
        raise NotImplementedError("abstract base")
    def is_positive(self):
        """ returns whether self is positive """
        raise NotImplementedError("abstract base")

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
```

Figure 1. A simple object model

If the function is executed using the tracing JIT, with `y` being a `BoxedInteger`, the produced trace looks like Figure 2.3. The operations in the trace are shown indented to correspond to the stack level of the function that contains the traced operation. The trace also 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 (corresponding to method calls).

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.[3] 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.

XXX simplify traces a bit more

[3] `guard_class` performs a precise class check, not checking for subclasses

```
# 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

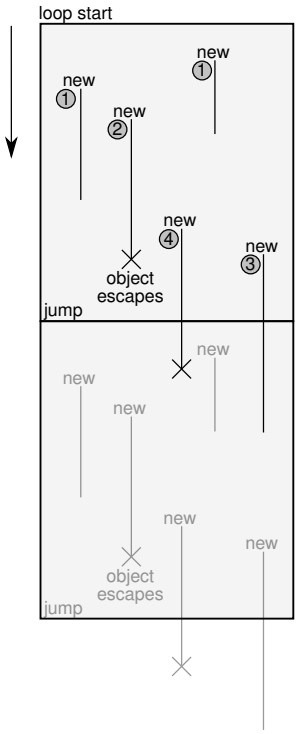

Figure 3. Object Lifetimes in a Trace

In the next section, we will see how this can be improved upon, using partial evaluation. XXX

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.

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.[4]

The objects that are allocated in the example trace in Figure 2.3 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. We will look at objects in category 3 in Section 6.

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.

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`, and that the `intval` field contains i_4, the one associated with p_6 would know that its `intval` field contains the constant -100.

The following operations, that use 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`

[4] In theory, the approach of Section 6 works also for objects that live for exactly $n > 1$ iterations and then don't escape, but we expect this to be a very rare case, so we do not handle it.

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 objects 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 2.3, 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 optimizers 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) 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 4.1.

The optimized trace contains only two allocations, instead of the original five, and only three `guard_class` operations, from the original seven.

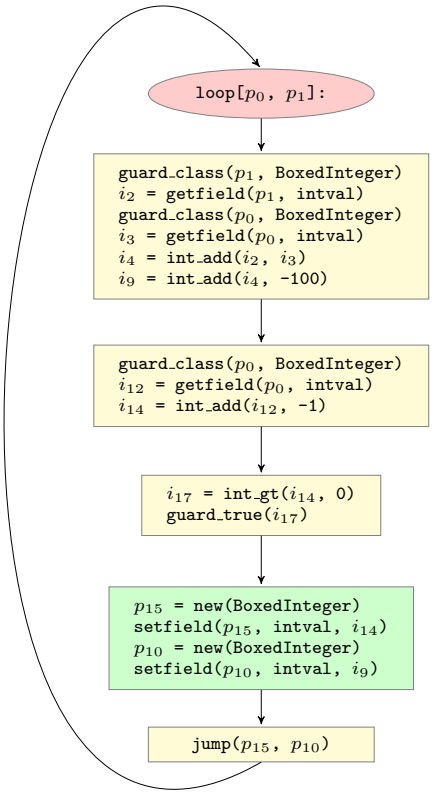

Figure 4. Resulting Trace After Allocation Removal

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` (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 immediately initialized to a real location, 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 and a heap, 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 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 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 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

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

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

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

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

Object Domains:

$u, v, w \in V$	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

is empty if the operation can be optimized away (which hopefully happens often).

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 to variables in the optimized trace (which are 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.

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^*. An 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 that nothing is known about the variables, which means the operation has to be residualized.

If the argument u 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 emitted.

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 is updated. Otherwise the `set` operation needs to be emitted. This needs to be done carefully, because the new value for the field stored in the variable u could itself be static, in which case it needs to be lifted first.

I 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 put into the residual trace. 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 that looks like 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

XXX algorithm is linear in the length of the trace

XXX Category 2 The optimization of Section 4 deals with them too: the `new` that creates them and the field accesses are deferred, until the point where the object escapes.

6. Allocation Removal Across Loop Boundaries

In the last sections we described how partial evaluation can be used to remove many of the allocations of short-lived objects and many of the type dispatches that are present in a non-optimized trace. In this section we will improve the optimization to also handle more cases.

The optimization of the last section considered the passing of an object along a jump to be equivalent to escaping. It was thus treating objects in category 3 and 4 like those in category 2.

The improved optimization described in this section will make it possible to deal better with objects in category 3 and 4. This will have two consequences: on the one hand, more allocations are removed from the trace (which is clearly good). As a side-effect of this, the traces will also be type-specialized.

6.1 Optimizing Across the Jump

[5]

Let's look at the final trace obtained in Section 4 for the example loop. The final trace (Figure 4.1) was much better than the original one, because many allocations were removed from it. However, it also still contained allocations.

[5] This section is a bit science-fictiony. The algorithm that PyPy currently uses is significantly more complex and much harder than the one that is described here. The resulting behaviour is very similar, however, so we will use the simpler version (and we might switch to that at some point in the actual implementation).

new

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

get

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

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

set

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

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

guard

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

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

lifting

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

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

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

Object Domains:

$u, v, w \in V$	variables in trace
$u^*, v^*, w^* \in V^*$	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

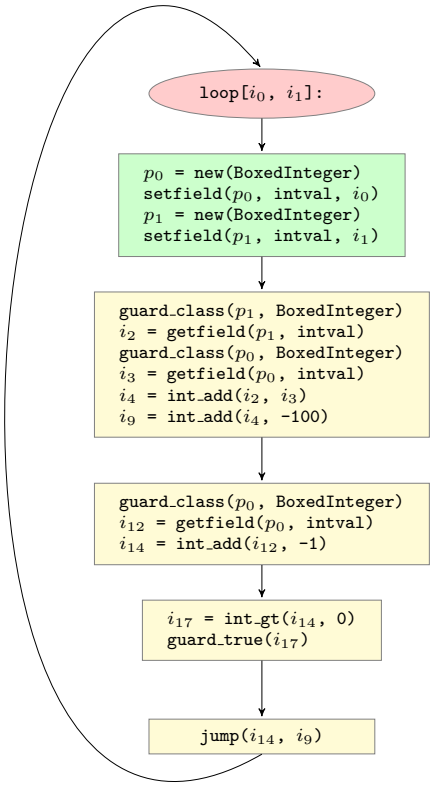

Figure 7. Shifting the Jump

The two new `BoxedIntegers` stored in p_{15} and p_{10} are passed into the next iteration of the loop. The next iteration will check that they are indeed `BoxedIntegers`, read their `intval` fields and then not use them any more. Thus those instances are in category 3.

In its current state the loop allocates two `BoxedIntegers` at the end of every iteration, that then die very quickly in the next iteration. In addition, the type checks at the start of the loop are superfluous, at least after the first iteration.

The reason why we cannot optimize the remaining allocations away is because their lifetime crosses the jump. To improve the situation, a little trick is needed. The trace in Figure 4.1 represents a loop, i.e. the jump at the end jumps to the beginning. Where in the loop the jump occurs is arbitrary, since the loop can only be left via failing guards anyway. Therefore it does not change the semantics of the loop to put the jump at another point into the trace and we can move the `jump` operation just above the allocation of the objects that appear in the current `jump`. This needs some care, because the arguments to `jump` are all currently live variables, thus they need to be adapted.

If we do that for our example trace, the trace looks like in Figure 6.1.

Now the lifetime of the remaining allocations no longer crosses the jump, and we can run our partial evaluation a second time, to get the trace in Figure 6.2.

This result is now really good. The code performs the same operations than the original code, but using direct CPU arithmetic and no boxing, as opposed to the original version which used dynamic dispatching and boxing.

Looking at the final trace it is also completely clear that specialization has happened. The trace corresponds to the situation in which the trace was originally recorded, which happened to

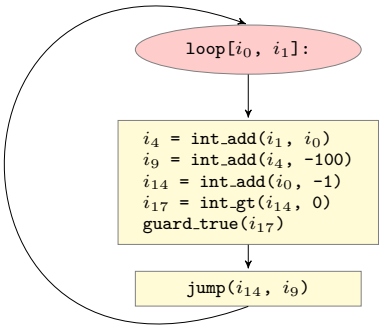

Figure 8. Removing Allocations a Second Time

be a loop where `BoxedIntegers` were used. The now resulting loop does not refer to the `BoxedInteger` class at all any more, but it still has the same behaviour. If the original loop had used `BoxedFloats`, the final loop would use `float_*` operations everywhere instead (or even be very different, if the object model had more different classes).

6.2 Entering the Loop

The approach of placing the `jump` at some other point in the loop leads to one additional complication that we glossed over so far. The beginning of the original loop corresponds to a point in the original program, namely the `while` loop in the function `f` from the last section.

Now recall that in a VM that uses a tracing JIT, all programs start by being interpreted. This means that when `f` is executed by the interpreter, it is easy to go from the interpreter to the first version of the compiled loop. After the `jump` is moved and the escape analysis optimization is applied a second time, this is no longer easily possible. In particular, the new loop expects two integers as input arguments, while the old one expected two instances.

To make it possible to enter the loop directly from the intepreter, there needs to be some additional code that enters the loop by taking as input arguments what is available to the interpreter, i.e. two instances. This additional code corresponds to one iteration of the loop, which is thus peeled off [citation needed], see Figure 6.2.

XXX optimization particularly effective for chains of operations

6.3 Summary

The optimization described in this section can be used to optimize away allocations in category 3 and improve allocations in category 4, by deferring them until they are no longer avoidable. A side-effect of these optimizations is also that the optimized loops are specialized for the types of the variables that are used inside them.

7. Supporting Techniques

7.1 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

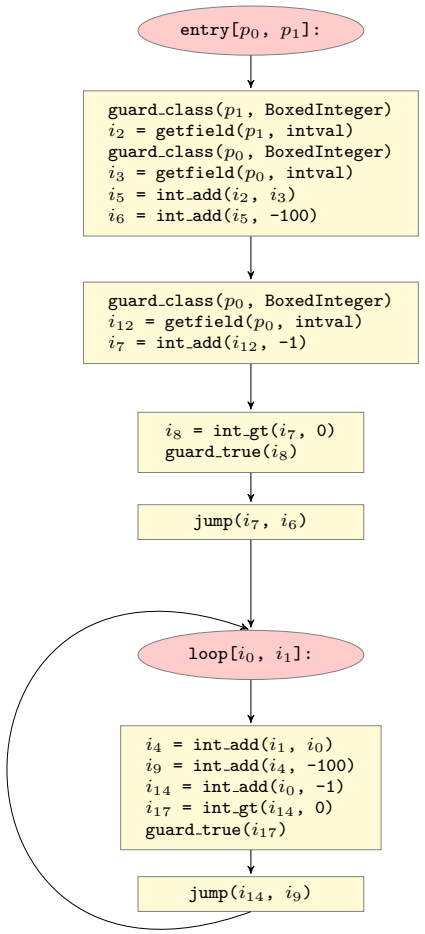

Figure 9. A Way to Enter the Loop From the Interpreter

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 [citation needed], the only novelty in our approach lays in its generality, because most other JITs are just specifically written for one particular language.

XXX store sinking
do we need this?

8. Evaluation

Benchmarks from the Computer Language Benchmark Game are: fannkuch, nbody, meteor-contest, spectral-norm.

crypto_pyaes: AES implementation. **django**: The templating engine of the Django web framework[6]. **go**: A Monte-Carlo Go AI[7]. **html5lib**: HTML5 parser **pyflate-fast**: BZ2 decoder **raytrace-simple**: ray tracer **richards**: The Richards benchmark [citation needed] **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].

9. Related Work

10. Conclusions

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] 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.

[4] 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.

[5] C. F. Bolz, M. Leuschel, and D. Schneider. Towards a jitting VM for prolog execution. Hagenberg, Austria, 2010. accepted for publication.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

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

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

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