Call-Target Agnostic Keyword Arguments

Maple Ong, Chris Seaton
Method arguments in Ruby
Positional arguments

```python
def foo(a, b):
    a + b
end
foo(1, 2)
```

Keyword arguments

```python
def foo(a:, b:)
    a + b
end
foo(a: 1, b: 2)
```

Combination of both

```python
def foo(a, b:)
    a + b
end
foo(1, b: 2)
```
def foo(**args)
    args[:a] + args[:b]
end

foo(**{a: 1, b: 2})
Generic keyword implementation
Generic keyword argument handling

- Keyword arguments wrapped into generic representation - a full Ruby hash object that is heap allocated
- Pushed onto the stack and receiver is looked up
- Dispatched to the call-target
- Call-target looks up each argument in the Ruby hash
Big reason why this is bad

Ruby Hash is allocated here...

...but the compilation boundary is in between the allocation and the use for non-inlined cases! Graal’s excellent optimisations don’t apply.

...and immediately consumed here, never to be used again...

def foo(a:, b:)
  a + b
end

foo(a: 1, b: 2)
Call-target-specific keyword implementation
Call-target-specific Method Arguments

Fabio Niephaus, Matthias Springer, Tim Felgentreff, Tobias Pape, Robert Hirschfeld

Software Architecture Group, Hasso Plattner Institute, University of Potsdam
{fabio.niephaus, matthias.springer}@student.hpi.uni-potsdam.de
{tim.felgentreff, tobias.pape, hirschfeld}@hpi.uni-potsdam.de

Abstract
Most efficient implementations of dynamically-typed programming languages use polymorphic inline caches to determine the target of polymorphic method calls, making method lookups more efficient. In some programming languages, parameters specified in method signatures can differ from arguments passed at call sites. However, arguments are typically specific to call sites, so they have to be converted within target methods. We propose call-target-specific method arguments for dynamically-typed languages, effectively making argument handling part of polymorphic inline cache entries. We implemented this concept in JRuby using the Truffle framework in order to make keyword arguments more efficient. Microbenchmarks confirm that our implementation makes keyword argument passing in JRuby more than twice as fast.

Categories and Subject Descriptors D.1.5 [Programming Techniques]: Object-Oriented Programming; D.3.4 [Programming Languages]: Processors—code generation, optimization

Keywords PIC, Method Arguments, Named Arguments, JRuby

1. Introduction

2. Example: Ruby Keyword Arguments
Keyword arguments (named arguments) in Ruby will serve as a running example in the remainder of this paper, but other constructs [12] such as variable-sized argument lists with a rest argument are amenable to our approach. The usage of keyword arguments is widely spread in Ruby: for instance, libraries like ActiveRecord typically pass options arguments as keyword arguments [3]. They are also useful for designing domain-specific languages [5]. Ruby 2.0 introduced a more compact syntax for keyword arguments (Listing 1), in addition to the old syntax.

```
def A.foo(a:, b:)
a + b
end

def B.foo(b:, a:)
a + b
end

def C.foo(a:, **kwargs)
a + kwargs[:b]
end
```
def foo(a, b)
    a - b
end

def bar(b, a)
    b - a
end

call_target = ...

if call_target == foo
    args = [a, b]
elsif call_target == bar
    args = [b, a]
else
    deopt
end

call_target.call(*args)
Niephaus et al., Call-target-specific Method Arguments

Figure 1: Polymorphic inline cache for method dispatch.
static inline int

args_setup_kw_parameters_lookup(const ID key, VALUE *ptr, const VALUE *const passed_keywords, VALUE *passed_values, const int passed_keyword_len)
{
    int i;
    const VALUE keyname = ID2SYM(key);

    for (i=0; i<passed_keyword_len; i++) {
        if (keyname == passed_keywords[i]) {
            *ptr = passed_values[i];
            passed_values[i] = Qundef;
            return TRUE;
        }
    }

    return FALSE;
}
Our hypothesis
Ruby can be polymorphic

...which means call-target-specific approaches may not apply

class A
def foo(a, b)
  a + b
end
end

class B
def foo(a, b)
  a - b
end
end

klass = rand(2) == 0 ? A.new : B.new
klass.foo(1, 2)
Call-target-specific implementation doesn’t work well with Ruby

- Simpler for call-target to receive arguments but more complex on the call-site
- And there are many more call-sites than call-targets
- Does not work well with polymorphism
- Does not work well with metaprogramming
Other considerations

- Ruby Hash representation of arguments will fail escape-analysis in non-inlined cases, or cases with a large number of keyword arguments
- Non-inline performance of keyword arguments has an overhead in CRuby and TruffleRuby
- Keyword arguments inherently straddle a compilation boundary
- Therefore requires more creativity to solve than conventional optimisations
What is the big idea 🧠
call-sites >> call-targets

Call-sites to send arguments in any format and for call-target to dynamically adapt to the argument it receives
Call-target-agnostic Method Argument Handling

- Argument values flattened and paired with a static descriptor
- Both are pushed onto the stack and receiver is looked up
- Dispatched to the call-target
- Call-target uses the descriptor to unpack arguments based on the index
How our idea works in theory
def foo(a, b)
    a + b
end

foo(1, 2)
def foo(a:, b:):
a + b
end

foo(a: 1, b: 2)
def foo(kwargs)
    kwargs[:a] + kwargs[:b]
end

foo({a: 1, b: 2})
def foo(keywords, values)
    values[keywords.index_of(:a)] + values[keywords.index_of(:b)]
end

foo([[:a, :b], [1, 2]])
```python
def foo(keywords, values)
    if keywords == [':a', ':b']  # pointer comparison
        values[0] + values[1]
    else
        deopt
    end
end

foo([':a', ':b'], [1, 2])
```
def foo(keywords, values)
    if keywords == [:a, :b]  # pointer comparison
        values[0] + values[1]
    elsif keywords == [:b, :a]  # pointer comparison
        values[1] + values[2]
    else
        deopt
    end
end

foo([:a, :b], [1, 2])
foo([:b, :a], [2, 1])
def foo(keywords, values)
    if keywords == []:  # pointer comparison
        values[0] + values[1]
    else:
        deopt
    end
end

# if call-target only sees []
# the check is removed
def foo(keywords, values)
    values[0] + values[1]
end

foo([], [1, 2])
foo([], [2, 1])
How our idea works in practice
@Specialization
protected void empty(EmptyKeywordDescriptor descriptor, Object[] values) {
    // no keyword arguments
}

@ExplodeLoop
@Specialization(guards = "descriptor == cachedDescriptor")
protected void cached(EmptyKeywordDescriptor descriptor, Object[] values,
    @Cached("descriptor") NonEmptyKeywordDescriptor cachedDescriptor,
    @Cached(value = "getSlots(cachedDescriptor)") WriteFrameSlotNode[] descriptorSlots) {
    for (int n = 0; n < cachedDescriptor.getLength(); n++) {
        final WriteFrameSlotNode frameSlot = descriptorSlots[n];
        frameSlot.write(values[n]);
    }
}
An Object Storage Model for the Truffle Language Implementation Framework

Andreas Woehr*, Christian Wirth†, Daniele Bonetta‡, Chris Seaton‡, Christian Humer*
Hanspeter Mössenböck*†

*Institute for System Software, Johannes Kepler University Linz, Austria
†Oracle Labs
{woehr, christian.humer, moessenboeck}@ssw.jku.at, {christian.wirth, daniele.bonetta, chris.seaton}@oracle.com

Abstract
Truffle is a Java-based framework for developing high-performance language runtimes. Language implementers aiming at developing new runtimes have to design all the runtime mechanisms for managing dynamically typed objects from scratch. This not only leads to potential code duplication, but also impacts the actual time needed to develop a fully-fledged runtime.

In this paper we address this issue by introducing a common object storage model (OSM) for Truffle that can be used by language implementers to develop new runtimes. The OSM is generic, language-agnostic, and portable, as it can be used to implement a great variety of dynamic languages. It is extensible, featuring built-in support for custom extension mechanisms. It is also high-performance, as it is designed to benefit from the optimizing compiler in the Truffle framework. Our initial evaluation indicates that the Truffle OSM can be used to implement high-performance language runtimes, with no performance overhead when compared to language-specific solutions.

Categories and Subject Descriptors D.3.4 [Programming Languages]: Processors—Run-time environments, Optimization

General Terms Algorithms, Languages, Performance

Keywords Dynamic languages, virtual machine, language implementation, optimization, Java, JavaScript, Ruby, Truffle

1. Introduction
Truffle (2015) is a framework for developing runtimes for several Truffle-based implementations for dynamic languages exist, including JavaScript, Ruby, Python, Smalltalk, and R. All of the existing implementations offer very competitive performance when compared to other state-of-the-art implementations, and have the notable characteristics of being developed in pure Java (in contrast to native runtimes that are usually written in C/C++).

To further sustain and widen the adoption of Truffle as a common Java-based platform for language implementation, Truffle offers a number of shared APIs that language implementers can use to optimize the AST interpreter in order to produce even more optimized machine code. In order to obtain high performance, however, there has still been one core component that the Truffle platform did not offer to language implementers, and that had to be implemented manually. This core component is the object storage model, that is, the runtime support for implementing dynamic objects. Indeed, language implementers relying on the Truffle platform have to implement their own language-specific model for representing objects, and then have to optimize the language runtime accordingly in order to optimize the AST interpreter for the characteristics of a certain language’s object model. Requiring language implementers to develop the object storage model of their new language from scratch is not only a waste of resources, but could also lead to questionable software engineering practices such as code duplication and non-modular design.

With the goal of solving the above limitation of the Truffle framework and with the aim of supporting language developers with a richer shared infrastructure, this paper introduces a new, language-independent, object storage model (OSM) for Tru-
arguments = {a: 1, b: 2, c: 3}

static_descriptor = [:a, :b, :c]

dynamic_data = [1, 2, 3]
Why this is an interesting design space?

- Method arguments handling will fundamentally straddle a compilation unit, unless the call-site is inlined
- Therefore, Graal’s typical optimizations does not apply
Still Playing to Graal’s Strength

- Relies on Graal and Truffle’s ability to create efficient inline caches on arbitrary guards
- Dynamic optimization results in specialized, compact code
- Fallbacks are handled by interpreter
What it achieves
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Implementation</th>
<th>Compilation Time</th>
<th>AST</th>
<th>IR</th>
<th>Code size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Long Caller</td>
<td>Control</td>
<td>1299 (414+885) ms</td>
<td>65</td>
<td>147/1728</td>
<td>6994</td>
</tr>
<tr>
<td></td>
<td>Call-target-agnostic</td>
<td>468 (289+179) ms</td>
<td>65</td>
<td>120/230</td>
<td>1078</td>
</tr>
<tr>
<td>Long Callee</td>
<td>Control</td>
<td>1056 (106+951) ms</td>
<td>137</td>
<td>711/1523</td>
<td>5970</td>
</tr>
<tr>
<td></td>
<td>Call-target-agnostic</td>
<td>266 (128+138) ms</td>
<td>72</td>
<td>92/152</td>
<td>570</td>
</tr>
<tr>
<td>Short Caller</td>
<td>Control</td>
<td>699 (312+388) ms</td>
<td>28</td>
<td>98/463</td>
<td>1714</td>
</tr>
<tr>
<td></td>
<td>Call-target-agnostic</td>
<td>490 (281+210) ms</td>
<td>29</td>
<td>90/187</td>
<td>754</td>
</tr>
<tr>
<td>Short Callee</td>
<td>Control</td>
<td>347 (142+205) ms</td>
<td>35</td>
<td>159/299</td>
<td>1158</td>
</tr>
<tr>
<td></td>
<td>Call-target-agnostic</td>
<td>242 (116+126) ms</td>
<td>24</td>
<td>44/98</td>
<td>422</td>
</tr>
</tbody>
</table>
Conclusion
Conclusion

- Ruby keyword arguments are logically very expensive
  - Pass a Ruby hash object of keywords and values and look up the values you want
- Previous published work tackled at the call-site this by putting arguments into a standard order for the call-target
  - But this requires knowing the call-target, and it requires extra work at the call-site
- Our hypothesis is that there are many more call-sites than call-targets, so it makes sense to put the work at the call-target
- Ruby’s idiomatic use also often means you may not know the call-target at a given call-site
- Therefore we instead have the call-site send a description of the keyword arguments, and separately their values, and have the call-target inline cache against the description