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 in 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. Micro-benchmarks 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

Dynamically-typed object-oriented programming languages typically use polymorphic inline caches [6] during method dispatch to find the corresponding target method quickly: a small number of class types is stored along with method pointers at every call site. In some programming languages, however, different call targets that are cached by the same call site can require different arguments: Ruby keyword arguments will serve as a running example in this paper and are introduced in Section 2.

The main contribution of this paper are *call-target-specific method arguments*: instead of converting arguments into a call-site-specific format and sharing them among all call targets, our approach aims to make method arguments call-target-specific. Based on the polymorphic type of the receiver, the call site directly converts the arguments to the method-specific format. Call-target-specific method arguments were designed for dynamically-typed programming languages; a different idea has been proposed and implemented for statically-typed programming languages and is presented briefly in Section 5.

ICOOOLPS '15, July 6, 2015, Prague, Czech Republic.

ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

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 wide-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.

```
1  def A.foo(a:, b:)
2    a + b
3  end
4
5  def B.foo(b:, a:)
6    a + b
7  end
8
9  def C.foo(a:, **kwargs)
10   a + kwargs[:b]
11 end
12
13 object.foo(a: 1, b: 2)
14 object.foo({a: 1, b: 2})
```

Listing 1: Example usage of keyword arguments in Ruby.

Keyword arguments are named arguments. The order in which they are passed in a method call does not matter. Excess keyword arguments are available as a dictionary ("hash" in Ruby) in the target method's body (`**kwargs` in line 9) or cause an argument error if no such *rest keyword parameter* is defined in the method's signature.

Whenever keyword arguments are passed at a call site, both MRI[1] and JRuby store all keyword arguments in a dictionary and use that dictionary as the last argument. Whenever a keyword argument is present in a method's signature, the method checks if the last argument is a dictionary, extracts the corresponding keyword argument from the dictionary and stores it in a temporary variable.

When executing line 13, Ruby first creates a dictionary containing all keyword arguments ("representation that is compatible with all call targets") and then extracts keyword arguments mentioned in the method signature again ("method-specific representation").

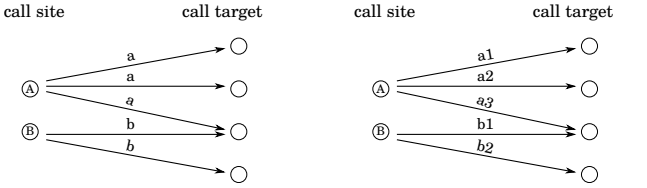
3. Call-target-specific Arguments

In this section, we explain how our solution is different from existing programming language implementations on a more abstract level. A language implementation typically performs the following steps when calling a method.

[1] MRI (Matz's Ruby Interpreter) is the Ruby reference implementation.

1. Convert arguments into a (generic) representation which is compatible with all call targets (e.g. create a dictionary containing all keyword arguments).
2. Push arguments on stack.
3. Look target method up (possibly using PIC).
4. Dispatch to target method.
5. Convert arguments to method-specific representation (e.g. extract keyword arguments mentioned in the method signature from the dictionary).
6. Execute method body.

Figure 1 shows polymorphic inline caches for call sites A and B with call-site-specific method arguments (a) and call-target-specific method arguments (b). In the former case, the exact same generic arguments are passed to all target methods for a specific call site. In the latter case, different arguments can be passed to target methods.



(a) Without call-target-specific arguments

(b) Without call-target-specific arguments

Figure 1: Polymorphic inline cache for method dispatch.

Although every call site along with its arguments is represented by a single statement in the source code (e.g. Listing 1, line 13), the execution environment might decide to pass different arguments for every polymorphic target type.

Polymorphic inline caches avoid unnecessary method lookups by caching call target methods for receiver types. Call-target-specific method arguments make argument handling more efficient by avoiding generic argument representations.

4. Implementation

In this section, we present our proof-of-concept implementation in JRuby. JRuby is an implementation of the Ruby programming language in Java and uses Truffle, an AST interpreter framework for AST node rewriting and partial evaluation (inlining) [10, 11]. Truffle runs on top of the Graal virtual machine [9], a modified HotSpot virtual machine, that provides runtime feedback to Truffle and can compile and install new optimized code on the fly.

4.1 JRuby Keyword Arguments

Figure 2 shows JRuby's implementation of polymorphic inline caches (*type decision chains*) for a call site. Initially, for the `CallDispatchHeadNode` no target method is cached so far, so it has a reference to an `UnresolvedDispatchNode` object. Upon successful method lookup, this reference is replaced by a reference to a `CachedDispatchNode` object, representing an entry in the polymorphic inline cache. That object has a reference to the next cache entry, which is initially an `UncachedDispatchNode` object but can be replaced by another cache entry, forming a linked list of cache entries. This list of cache entries is replaced by an `UncachedDispatchNode` object once the length of the list exceeds a threshold value, falling back to regular method lookup.

The original JRuby implementation stores the array of AST nodes for method arguments in `RubyCallNode`, which is specific to a certain call site. Our optimization stores method arguments in `DispatchNode` which is specific to a certain call target. Upon creation of PIC entries, our implementation analyzes the target method and stores AST nodes in the array of arguments in such a way that the i^{th} argument in the method signature corresponds to the i^{th} argument in the array, making it easy to load arguments within the target method without having to extract arguments from a dictionary.

Example Consider the case that line 13 in Listing 1 is executed multiple times, but `object` alternates between instances of `A`, `B`, and `C`. Our implementation generates three different argument arrays that are specific to one of the three `foo` methods[2].

- `A.foo: [1, 2]`
- `B.foo: [2, 1]`
- `C.foo: [1, {:b => 2}]`

Pitfalls and Implementation Details In Ruby, keyword arguments can be passed explicitly as named arguments (Listing 1, line 13) and implicitly using an already existing dictionary object (Listing 2, line 14). The latter case cannot be optimized by expanding dictionaries. Therefore, the original implementation should stay in place for this case.

Target methods must be able to distinguish whether the arguments of the current invocation are optimized (expanded) or not. We use a special marker object (Figure 3) at the end of the arguments array to denote that a call is optimized and keyword arguments can be read from the array directly without looking them up in a dictionary.

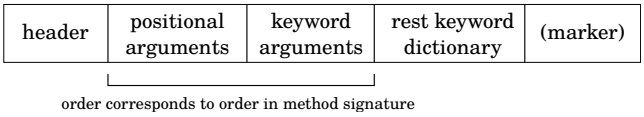


Figure 3: Ruby arguments array during method dispatch in JRuby.

Since we might rearrange AST nodes in the arguments array, the evaluation order of arguments might have changed. We solve this problem by evaluating arguments in their original order, storing their values into newly-created temporary variables, and passing AST nodes reading the temporary variables, possibly in a different order.

4.2 Benchmarks

We ran benchmarks (see Appendix) on our optimized JRuby implementation with Truffle on an ordinary HotSpot VM (no Graal). The benchmarks were run on a MacBook Pro with an i7-2720QM CPU and 16 GB RAM. The benchmarking code and our implementation, along with an install script, is also available for download[3].

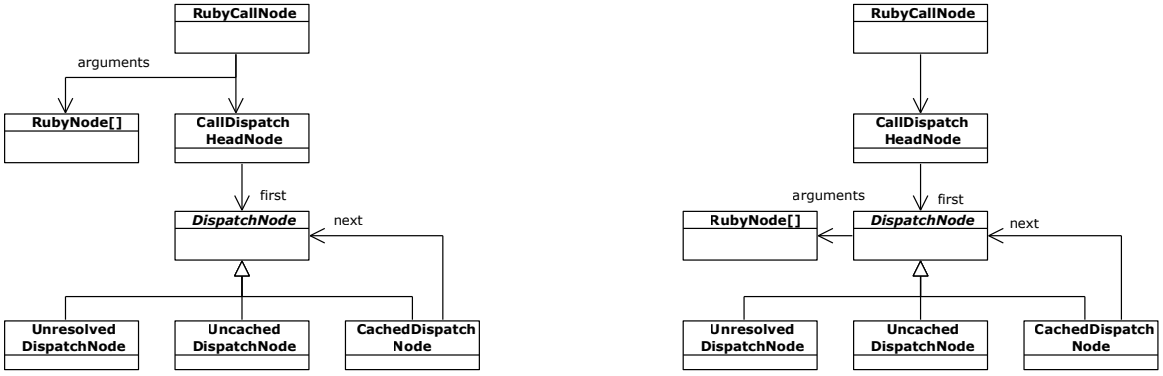
For every benchmark, we compare two implementations: the original, unoptimized implementation of JRuby (*call-site-specific*); and the optimized call-target-specific implementation. In the latter case, a separate AST subtree is stored with every PIC entry[4] (an AST subtree generating the array of arguments passed to the target method is stored for every PIC entry).

We benchmarked calling a method on object `obj` with 10 keyword arguments (Figure 4). In (a) and (d), `obj` is always the same object, whereas in (b) and (c), `obj` alternates between two objects,

[2] For better readability, we use Ruby syntax. Since the arguments array is part of the AST, all objects should be AST nodes (fixnum literal nodes, symbol literal nodes and hash literal nodes). Marker nodes are omitted.

[3] `https://github.com/HPI-SWA-Lab/TargetSpecific-ICOOOLPS`.

[4] This is different from Truffle AST node rewriting because multiple AST subtrees can be stored for a call site at the same time.



(a) Call-site-specific (b) Call-target-specific

Figure 2: Polymorphic inline caching and argument AST nodes in JRuby with Truffle.

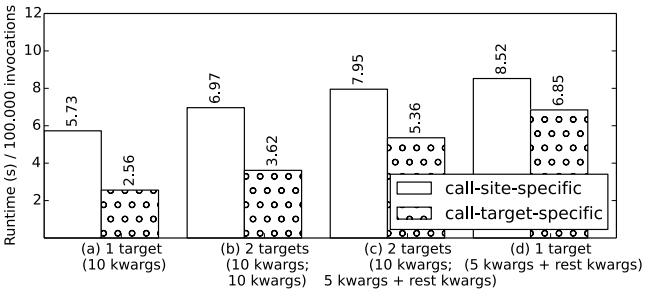


Figure 4: Benchmark comparing call-site-specific and call-target-specific method invocation.

whose methods have a different signature. In (a) and (b), the target methods take 10 keyword arguments, whereas in (d), the target methods takes 5 keyword arguments and a rest keyword dictionary. In (c), one of the target methods takes 10 keyword arguments.

Reading a keyword argument from the rest keyword argument hash is slow. That is why (a) is fastest benchmark and (d) is the slowest benchmark.

5. Related Work

Named parameters are supported in a wide range of programming languages. For example, Scala supports named arguments which can be optional ("implicit"). Since Scala is a statically-typed programming language, the polymorphic receiver type and, therefore, the target method's signature is known, because methods overridden in subclasses must have the same arguments as the base method. During method application, a collection consisting of all positional arguments followed by a fixed permutation of all named arguments defined in the signature is passed [7]. Scala evaluates arguments in their original order, stores them in temporary variables, and passes these variables in a permutated order, just as our implementation does. For every optional argument, there is a *default method* (automatically generated in the receiver's class) returning the value of the optional argument. When a method is called with optional arguments missing, the compiler generates code that first calls the corresponding default methods and then passes their return values together with the provided arguments. This is an elegant approach to make argument handling efficient, however, it cannot be applied in dynamically-typed programming languages, because the receiver type of a method call is unknown before dispatching the method call; therefore, the execution environment cannot know if a default argument is missing before looking up the receiver type.

MagLev is an implementation of the Ruby programming language on top of the GemStone/S virtual machine (Smalltalk). For every Ruby method, MagLev generates a set of *bridge methods* specific to a certain number of arguments and splat arguments. When calling a Ruby method, MagLev calls the corresponding bridge method which might provide default arguments and prepare a splat argument array, and calls the actual implementation. For example, if `foo(a = 1, *args)` is called via `foo(1, 2, 3)`, MagLev calls the bridge method `foo#3__` specific to three arguments which wraps the last two arguments in an array and calls the actual implementation [8]. If we extend this idea to keyword arguments, MagLev could also create bridge methods specific to keyword argument labels: whenever a method call with keyword arguments is compiled, MagLev could add a corresponding bridge method to all classes having a method with that name; e. g. compiling `foo(1, b: 2, a: 1)` could generate bridge methods `foo#1__a_b` (one positional argument, keyword arguments `a` and `b`[5]). This would avoid creating and expanding dictionaries when using keyword arguments, but it could also increase the number of methods significantly: in contrast to making arguments call-target-specific, this approach would add call-site-specific wrapper methods.

Gil and Lenza propose an extension of the Java programming language to support keyword arguments and optional arguments [4]. Their implementation generates *auxiliary methods* for all valid calling patterns, which is similar to MagLev's concept of bridge methods. Auxiliary methods provide missing arguments and call the actual implementation with the full set of arguments.

In programming languages supporting multiple dispatch, the call target depends on all arguments (usually argument types) instead of just the receiver argument [1]. Multiple dispatch is used to provide entirely different method implementations for different argument types. Call-target-specific method arguments are, however, an optimization *under the hood*, and aim at making argument handling more efficient, instead of providing different semantics for different argument types.

6. Conclusion

We presented call-target-specific method arguments, making method arguments part of polymorphic inline cache entries. Our idea can be used to adapt call sites to do argument conversions a single time whenever a new entry is added to a polymorphic inline cache.

[5] Keyword arguments are sorted lexicographically.

When a method call is then executed and the receiver's type was found in the cache, the callee can directly start executing the method body without having to do argument conversions like finding named arguments inside a dictionary.

Future work might investigate how call-target-specific method arguments can be applied to other programming languages. For example, Python supports named arguments which are similar to Ruby's keyword arguments. However, it is unclear to what degree efficient implementations such as PyPy [2] will benefit from this optimization, since their tracing JIT compiler might already be able to improve the usage of named arguments. PyPy analyzes passed arguments and initializes local variables in stack frames by going through all named arguments (name-value pairs); this iteration step could be omitted at subsequent method calls when using call-target-specific method arguments.

Acknowledgments

We would like to thank Chris Seaton for his support with our JRuby implementation.

References

[1] S. Bansal. Multiple polymorphic arguments in single dispatch object oriented languages. In S. Ranka, A. Banerjee, K. Biswas, S. Dua, P. Mishra, R. Moona, S.-H. Poon, and C.-L. Wang, editors, *Contemporary Computing*, volume 95 of *Communications in Computer and Information Science*, pages 260–271. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-14824-8.

[2] C. F. Bolz, A. Cuni, M. Fijalkowski, 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*, ICOOOLPS '09, pages 18–25, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-541-3.

[3] L. Carlson and L. Richardson. *Ruby Cookbook, 2nd Edition*. O'Reilly Media, Inc., 2015. ISBN 978-1-4493-7371-9.

[4] J. Y. Gil and K. Lenza. Keyword- and default- parameters in java. *Journal of Object Technology*, 11(1):1:1–17, Apr. 2012. ISSN 1660-1769.

[5] S. Günther and T. Cleenewerck. Design principles for internal domain-specific languages: A pattern catalog illustrated by ruby. In *Proceedings of the 17th Conference on Pattern Languages of Programs*, PLOP '10, pages 3:1–3:35, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0107-7.

[6] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, pages 21–38, London, UK, UK, 1991. Springer-Verlag. ISBN 3-540-54262-0.

[7] L. Rytz and M. Odersky. Named and default arguments for polymorphic object-oriented languages: A discussion on the design implemented in the scala language. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2090–2095, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-639-7.

[8] M. Springer. Inter-language collaboration in an object-oriented virtual machine, 2013. Bachelor's thesis, Hasso Plattner Institute.

[9] C. Wimmer and T. Würthinger. Truffle: A self-optimizing runtime system. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 13–14, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1563-0.

[10] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-optimizing ast interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, pages 73–82, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1564-7.

[11] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 187–204, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2472-4.

[12] U. Zdun. Patterns of argument passing. In *Proceedings of the 4th Nordic Conference of Pattern Language of Programs (Viking-PLoP2005)*, pages 1–25, 2005.

A. Appendix

```
class A
 def foo_a(a:1, b:2, c:3, d:4, e:5, f:6, g:7, h:8, ↩
      i:9, j:10)
  a + b + c + d + e + f + g + h + i + j
 end

 def foo_b(a:1, b:2, c:3, d:4, e:5, f:6, g:7, h:8, ↩
      i:9, j:10)
  a + b + c + d + e + f + g + h + i + j
 end

 def foo_c(a:1, b:2, c:3, d:4, e:5, f:6, g:7, h:8, ↩
      i:9, j:10)
  a + b + c + d + e + f + g + h + i + j
 end
end

class B
 def foo_b(j:11, i:12, h:13, g:14, f:15, e:16, d:17, ↩
      c:18, b:19, a:20)
  a + b + c + d + e + f + g + h + i + j
 end

 def foo_c(j:11, i:12, h:13, g:14, f:15, **kwargs)
  kwargs[:a] + kwargs[:b] + kwargs[:c] + kwargs[:d] + ↩
       kwargs[:e] + f + g + h + i + j
 end

 def foo_d(j:11, i:12, h:13, g:14, f:15, **kwargs)
  kwargs[:a] + kwargs[:b] + kwargs[:c] + kwargs[:d] + ↩
       kwargs[:e] + f + g + h + i + j
 end
end

# Benchmark (a): one target A
obj.foo_a(a:1, b:2, c:3, d:4, e:5, f:6, g:7, h:8, i:9, ↩
     j:10)

# Benchmark (b): two targets A, B alternating
obj.foo_b(a:1, b:2, c:3, d:4, e:5, f:6, g:7, h:8, i:9, ↩
     j:10)

# Benchmark (c): two targets A, B alternating
obj.foo_c(a:1, b:2, c:3, d:4, e:5, f:6, g:7, h:8, i:9, ↩
     j:10)

# Benchmark (d): one target B
obj.foo_d(a:1, b:2, c:3, d:4, e:5, f:6, g:7, h:8, i:9, ↩
     j:10)
```

Listing 2: Simplified benchmarking code.