Supporting Objects in Run-time Bytecode Specialization

Reynald Affeldt † Hidehiko Masuhara ‡ Eijiro Sumii † Akinori Yonezawa †

†Department of Information Science, Graduate School of Science, University of Tokyo

‡Department of Graphics and Computer Science, Graduate School of Arts and Sciences, University of Tokyo

This paper describes a run-time specialization system for the Java language. One of the main difficulties of supporting the full Java language resides in a safe yet effective management of references to objects and arrays. This is because a specialization process may share references with the process that executes specialized code, and because side-effects to those objects by the specialization process could easily break the semantics of the original program. To cope with this difficulty, we propose a set of requirements and assumptions that ensures correct run-time specializations. Based on them, we design and implement a run-time specialization system for Java. Our preliminary experiments reveal, for instance, approximately 20-25% speed-up factor for a ray-tracing application.

1 Introduction

Partial evaluation (or specialization, for short) is a program transformation technique that performs constant propagation and expression unfolding in a program given partial knowledge of its arguments. This optimization is particularly effective in optimizing highly parameterized programs running in invariant contexts. When performed at run-time, specialization has the additional advantage to enable optimization with respect to infrequently-changing run-time values.

Run-time specialization has been experimented for a large number of programming languages (e.g., [3], [5]). There are also numerous studies about both practical and theoretical aspects of compile-time specialization of imperative and object-oriented languages (e.g., [7]). However, few systems fully support both run-time specialization and object-orientation.

In fact, we found it is non-trivial to design efficient and correct run-time specialization for an object-oriented language. Because specialization is done during a run of an application program, specializers and specialized methods are tightly coupled by the use of the heap. This introduces new correctness constraints but also new optimization opportunities. Most techniques for run-time specialization merely support immutable data structures or put the user in charge of writing a number of annotations to achieve correct and effective specialization of mutable data structures.

This paper describes a run-time specialization system for a typical object-oriented language, namely the Java language. In Sect. 2, we discuss why run-time specialization of Java is both challenging and highly beneficial. In Sect. 3, we give an overview of our implementation. In Sect. 4, we show how we support references. In Sect. 5, we discuss the treatment of local objects. In Sect. 6, we present results of our preliminary performance.

2 Motivations

We believe that run-time specialization of object-oriented languages may be highly beneficial because:

- it makes use of run-time invariants, thus triggering more optimizations,
- it resolves virtual dispatches that cannot be eliminated by traditional static analyses, and
- it benefits from the fact that specializers and specialized methods share a common heap.

The last point, at the same time, complicates correct specialization. This can be explained by the following example that manipulates a one-dimensional point:

```
1  Class Point {
2          int x = 0;
3          // f performs some heavy computation
4          void update (int a) { x = f (x, a); }
5          static Point make (int s, int d) {
6                  Point p = new Point ();
7                  p.update (s);
8                  p.update (d);
9                  p.update (s);
10                 return p;
11         }
12 }
```

Assume we specialize the method `make` with respect to `s`. The object construction at line 6 only depends on *static* arguments (their values are known), it can thus be performed at specialization-time. The constructed object is recorded in a global variable `_p`. The method call of line 7 is evaluated away because both the receiver object and the argument are static. At line 8, a method of object `p` is invoked with a *dynamic* argument (its value is unknown), entailing residualization of the method call. After that, the status of object `p` becomes unknown, resulting in residualization of the method call of line 9. Eventually, the specialized code becomes:

```
static Point make_spec (int d) {
        _p.update (d);  /* _p is the static point
                constructed during specialization */
        _p.update (42); // 42 is the value of s
        return _p;
}
```

In this specialized code, construction of an object and one of the heavy computation are specialized away by sharing objects through a global variable. However, this code is incorrect for the following reasons:

- First, if we call the specialized method twice, the second call will return an object in the wrong state. In fact, the specialized code implicitly assumes that `_p` is reinitialized, while it is not.

- Second, two invocations of the specialized method return the same reference, whereas the subject method originally returned a fresh reference at each run.

At first sight, the idea of reusing the local object constructed at specialization-time was tempting, in particular for a language like Java where a number of short-lived objects are extensively used; e.g., objects in classes `String`, `Rectangle`, and `Font`.

A conservative solution would be to turn object `p` into a dynamic one, but it would result in poor specialization. This is unsatisfactory for today's object-oriented programs that usually define methods for constructing and initializing objects (e.g., the factory design pattern).

We show in Sect. 5 how we preserve the correctness of the program transformation while enabling reuse of the objects constructed at specialization-time.

3 BCS Overview

ByteCode Specializer (BCS) is a run-time specializer for a subset of the Java Virtual Machine Language (JVML) [4]. The specialization is offline and uses code generators (*generating extensions*). Thus far, BCS has no support for references to objects and arrays. We implement our run-time specializer for the Java language as an extension of BCS. Note that, even though our examples are written in Java (or at least in pseudo-instructions that resemble Java), the underlying system actually still manipulates bytecodes.

We now present the three stages that compose the run-time specialization process. We use as a running example the method that computes in a ray tracer the intersection between a ray of light and a scene that is the closest to an observer:

```
Inter inter = ray.closestInter (observer, scene);
```

We specialize `closestInter` with respect to the static `observer` and `scene` objects, the `ray` object being dynamic.

3.1 Analysis Stage

The analysis stage takes place at compile-time and amounts to call BCS with the compiled version of the subject method `closestInter` and a binding-time specification distinguishing the static arguments from the dynamic ones.

The first step is to determine the shape of the Java virtual machine elements (stack, frame, heap) in the presence of references to objects and arrays. Next, type variables are attached to each stack entry, frame variable, heap slot, and bytecode instruction. Dependencies between those constructs are then instantiated. A set of typing rules covering the core JVML language (including bytecodes that manipulate objects) concisely expresses those dependencies. Intuitively, they state that a bytecode instruction is static if it solely depends on static arguments. Dependencies are eventually resolved. A code generator generator further translates the resulting annotated program into a code generator, that effectively performs the specialization.

In our example, the result of the analysis stage is the code generator `closestInter_gen`.

3.2 Specialization Stage

The specialization stage takes place at run-time and amounts to call the previously generated code generator `closestInter_gen` with the actual values of the static `observer` and `scene` objects.

The result of the specialization stage is the specialized method `closestInter_spec`[1].

3.3 Execution Stage

The execution stage consists in replacing the call to the generic method `closestInter` by a call to its specialized version `closestInter_spec` to which is passed the actual value of the dynamic `ray` object. Concretely:

```
Inter inter = ray.closestInter (observer, scene);
```

becomes

```
Inter inter = closestInter_spec (ray);
```

4 References and Objects Support

We call *static context* the set of actual values of the static arguments. If the target language has references, then the static context extend over all the heap slots that are reachable from them. We call *static heap* that part of the heap. Since a specialized code is generated for each static context, it may return a different result from the one of the original code when it is executed in a modified static context. To illustrate the assumptions on which BCS is based, we use the following example.

This program shows how specialization may take place in the control flow of an application:

```
Point p = new Point ();
p.update (s);
// specialize update with respect to p and s
update_spec = p.update_gen (s);
p.update (s);
```

If `update_gen` is to change the static heap by modifying `p`'s coordinate, the application will be disrupted in a rather unnatural way. Moreover, the correctness of specialization should require the execution of the specialized method to take place just after its specialization. In those conditions, it becomes difficult to extensively reuse the specialized method, which is an inconvenient for a system whose quality is partly determined by its amortization cost. Again, turning the object into dynamic solves the problem but results in poor specialization.

To enable an effective specialization, we allow at specialization-time static reads while prohibiting static writes.

The immediate consequence is the possibility for the specializer to residualize direct accesses to the static

[1] The actual implementation returns an instance of a class that implements the specialized method.

heap, thus imposing the specializer and the specialized code to run against the same static heap [2]. This requirement is fairly natural since it corresponds to the traditional correctness definition of partial evaluation.

4.1 No Static Heap Destruction

To guarantee the above property, we must prohibit static assignments through references to the static heap. In order to detect such assignments, we perform a side-effect analysis before the binding-time analysis.

More precisely, the binding-time analysis rule for assignments through references states that if the reference may point to the static heap, then (1) the assignment is given dynamic binding-time, and (2) all the referenced slots are marked as dynamic to disable subsequent static dereferences.

4.2 Persistent Static Heap

This assumption about the static heap requires the user to keep the static heap unchanged from the specialization stage to the execution stage. Although the current implementation has no automatic mechanism, a piece of guard code can also be automatically produced for that purpose.

A persistent static heap enables *reference lifting*. We say that a primitive value is lifted when it appears in a dynamic context. Concretely, it is residualized to a piece of code that yields its value at execution-time. In a compile-time specializer, references cannot be lifted because it is difficult before execution to represent them by a syntactic construct. In contrast, in a run-time specializer that ensures persistence of the static heap, a specialized method can safely refer to lifted references.

Since it it not possible in JVML to yield directly onto the stack the value of a reference, we need to save it at specialization-time in a table from which it will be retrieved by a piece of residual code at execution-time.

Given our ability to lift references, support for partially static objects is a trivial matter. This is actually an intended feature. Complex data structures are typical of object-oriented programs and our ability to handle partially static object is therefore critical to deliver an accurate specialization.

In the case of the example above, the specialized method written in pseudo-instructions becomes:

```
// p is the static point argument
// 7 is the value of p's coordinate
// 42 is the value of the static integer argument
// 1234 is the value returned by f (7, 42)
void update_spec () { p.x = 1234; }
```

4.3 Virtual Dispatching

Thanks to the knowledge of the run-time class of static arguments, BCS can evaluate away virtual dispatches. It is all the more effective since other optimizations like loop unrolling clear up method calls. In comparison, a class-hierarchy analysis occurring at compile-time is overly conservative.

By way of example, let us consider virtual dispatch call sites in our ray tracer. They appear in particular where the intersection between a ray of light and an object needs to be computed:

```
Inter inter = object.intersect (observer, ray);
```

The declared class of `object` is an abstract class whose subclasses include a `Plane` class and a `Sphere` class. When BCS runs into such a call site, the generated code generator is added the following method call:

```
Inter inter = object.intersect_gen (observer, ray);
```

and the classes `Plane` and `Sphere` are added code generators with the same signature. As a result, virtual dispatching is resolved at specialization-time by the Java virtual machine.

5 Local Object Support

Correct and efficient treatment of the construction of local objects is a difficult issue that requires us to reconcile two opposite requirements. On the one hand, escaping references to local objects must be guaranteed to be fresh. On the other hand, the binding-time analysis should annotate as many constructs as possible as static, including objects allocated in the target method.

Our solution is to perform at specialization-time all the object constructions that do not depend on any dynamic value. The corresponding local objects are considered static and the corresponding allocated objects, that we call *template objects*, are kept in the *specialization store*.

The specializer can use the template object to carry out static operations, no matter whether the local object escapes or not. As far as the specialized method is concerned, it uses the template object as a mould to instantiate new objects in the case of an escaping reference or as a place-holder to carry out dynamic computations when the reference does not escape.

5.1 Unique Identity Generation

In order to generate fresh identity for escaping local objects, we use a preliminary escape analysis. When there is a construction (carried out by the `new` operator) of an escaping object, the specializer first creates an object, registers the object in a table, and generates a `CLONE` pseudo-instruction. In the execution stage, the `CLONE` pseudo-instruction allocates a new object and initializes its instance variables by copying those of the object registered in the table.

We can now propose a correct specialized method for the example of Sect. 2:

```
static Point make_spec (int d) {
        Point p = CLONE _p; /* _p is the static point
                constructed during specialization */
        p.update (d);
        p.update (42);
        return p;
}
```

Although we cannot avoid the construction of the local object, `CLONE` saves one possibly time-consuming

`update` call, resulting in better specialization than making the local object dynamic.

Note that `CLONE` cannot be a straightforward shallow copy. When the to-be-`CLONE`d object is referencing other escaping local objects, they are also `CLONE`d so that objects created by cloning are eventually isomorphic to the objects created by the original code.

5.2 Specializing Initialization

Although it is not possible to avoid construction of escaping local objects, specialized methods can reuse template objects of non-escaping local objects.

For instance, let us consider a method that performs some arithmetic operations on complex numbers:

```
Complex f (Complex s, Complex d) {
        return s.plus(s).times(d);
}
```

We specialize it with respect to `s` whose value is $2+3i$. The specialized method written in pseudo-instructions becomes:

```
Complex f_spec (Complex d) {
        return _t.times(d); /* _t is the intermediate
                complex number 4 + 6i constructed
                during specialization */
}
```

Each instance of the specialized method `f_spec` reuses the same intermediate `Complex` object, whose dynamic allocation may be said to have been evaluated at specialization-time.

From the reference identity point of view, this is correct because non-escaping local objects are not involved in any equality test outside the subject method. Inside the specialized method, they can only be compared against dynamic references that are necessarily different (dynamic references cannot be aliases since this would have required escaping).

Some precautions must however be taken to guarantee data integrity. Whenever the template object is lifted inside the subject method, a mechanism should be added that reinitialize it to its specialization-time state before it can be reused. However, this is unnecessary in the other cases, independently of the fact that the object is immutable or not. Indeed, specialization residualizes the code that correspond to the object proper initialization (for more details, see the full version [1] of this paper).

This reusing is actually reminiscent of an optimization technique used in languages with automatic memory management which consists in recycling memory slots that are statically known to be unused.

6 Performance Measurements

We implemented the system by extending the second and the last author's BCS system, and compared the execution time of subject methods and of their specialized versions for two object-oriented applications specialized with BCS. We also measure the overhead due to the JIT compilation. Times are given in microseconds.

Our first example is an implementation of the power function that is specialized with respect to its exponent [7]. It is written in an object-oriented manner by using the Strategy design pattern:

VM		unspec.	spec.	speed-up
UltraSparcII	method	1.99	1.37	1.46
Sun JDK 1.3.1	JIT	40,039	181,237	
Pentium III	method	0.54	0.42	1.30
Sun JDK 1.3.1	JIT	12,849	27,392	
Pentium III	method	0.32	0.07	4.35
IBM JDK 1.3.0	JIT	48,921	54,081	

Our second example is the ray tracer discussed in Sect. 3 and Sect. 4.3:

VM		unspec.	spec.	speed-up
UltraSparcII	method	10.18	8.65	1.18
Sun JDK 1.3.1	JIT	196,055	200,485	
Pentium III	method	6.40	5.12	1.25
Sun JDK 1.3.1	JIT	115,241	104,031	
Pentium III	method	9.87	7.84	1.26
IBM JDK 1.3.0	JIT	208,341	557,194	

We get reasonable performance improvements. However, experiments of run-time specialization with other imperative languages seem to reach slightly better speed-up (e.g., [3]). We believe that this is due to the difference in the implementation of traditional compilation optimizations in Just-in-time compilers and also to the lack of optimizations in our system at analysis-time.

7 Conclusion and Future Work

This paper presented a run-time specialization system for the Java language with discussion on supporting references and objects. In particular, it correctly residualizes side-effects, handles partially static objects, and resolves statically determined virtual dispatches. Treatment of local objects has been given special attention so that it preserves reference identity while enabling safe reuse of the specialization store.

The next step is to further validate experimentally our implementation. A precise side-effect analysis that enables more static side-effects should also be devised.

References

[1] R. Affeldt. Supporting object-oriented features in run-time bytecode specialization. Master's thesis, University of Tokyo, Graduate School of Science, Department of Information Science, Sep 2001.

[2] K. Asai. Integrating partial evaluators into interpreters. In *Semantics, Applications and Implementation of Program Generation*, Lecture Notes in Computer Science. Springer-Verlag, Sep 2001. To appear.

[3] N. Fujinami. Automatic run-time code generation in C++. In *Scientific Computing in Object-oriented Parallel Environments*, volume 1343 of *Lecture Notes in Computer Science*, pages 9–16. Springer-Verlag, 1997.

[4] H. Masuhara and A. Yonezawa. A portable approach to dynamic optimization in run-time specialization. *Journal of New Generation Computing*, 20(1), Jan 2002.

[5] F. Noël, L. Hornof, C. Consel, and J. L. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. In *International Conference on Computer Languages*, pages 123–142. IEEE, May 1998.

[6] U. Schultz and C. Consel. Automatic program specialization for Java. Technical report, IRISA/INRIA, Rennes - LABRI, Nov 2000. Available at `http://www.daimi.au.dk/PB/551/PB-551.pdf`.