Supporting Objects in Run-Time Bytecode Specialization

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

†Department of Computer Science
Graduate School of Information Science and Technology
University of Tokyo

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

{affeldt,masuhara,sumii,yonezawa}@yl.is.s.u-tokyo.ac.jp

ABSTRACT

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 sound yet effective management of references to objects. This is because the specialization process may share references with the running application that executes the residual code, and because side-effects through those references by the specialization process could easily break the semantics of the running application. To cope with these difficulties, we elaborate requirements that ensure sound run-time specialization. Based on them, we design and implement a run-time specialization system for the Java language, which exhibits, for instance, approximately 20-25% speed-up factor for a ray-tracing application.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Code generation, Optimization, Run-time environments*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Partial evaluation*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Object-oriented constructs*

General Terms

Performance, Languages

Keywords

Specialization, Object-Oriented Paradigm, Partial Evaluation, Program Transformation

1. INTRODUCTION

Program specialization (or partial evaluation) is a technique that transforms a given program (called *subject* program) into an optimized program (called *residual* program) by assuming that some of the parameters of the program are particular constants (called *static* parameters). The residual program takes values of the remaining parameters (called *dynamic* parameters), and more efficiently computes the same result than the subject program does.

Recent progress in specialization techniques called *run-time specialization* makes specialization processes so efficient that they can be used at run-time [5, 7, 8, 9, 10, 12, 13, 15, 14, 16]. They thus offer more opportunities to specialize programs by using quasi-invariants that are only available at run-time, such as user's input and intermediate results of computation.

On the other hand, it is not easy to effectively optimize object-oriented programs by using existing specialization techniques, including run-time specialization techniques. This is because many object-oriented programs extensively use destructive updates and object identity (i.e., reference equality), which makes sound specialization difficult. As a result, existing specialization systems are either conservative (e.g., those that only specialize individual objects [7, 21]) or unsound without careful annotations [5, 9, 10, 16].

This paper presents a run-time specialization technique that soundly and effectively specializes object-oriented programs. For soundness, we define a set of requirements for sound specialization that apply to the specialization process, user processes and the residual code. Based on those requirements, our technique exploits the advantages of run-time specialization in two ways: (1) it can specialize programs with respect to objects constructed at run-time and (2) its heap-reusing mechanism allows the specialization process to construct objects so that the residual code can reuse them.

In addition to these advantages, the use of existing techniques such as partially static data, escape analysis (or region inference), and method inlining, effectively specializes object-oriented programs. In fact, our implementation, which is for the Java Virtual Machine language, effectively optimizes realistic object-oriented applications including ray-tracing rendering.

The rest of the paper is organized as follows. Sect. 2 discusses problems in run-time specialization of object-oriented programs. Sect. 3 overviews our run-time specialization system. Sect. 4 defines the requirements for sound specialization. Sect. 5 discusses issues on implementation. Sect. 6 shows how object-oriented programs can be specialized with the proposed mechanisms. Sect. 7 presents performance measurements of our implementation. We review related work in Sect. 8 and Sect. 9 concludes with a discussion on further research directions.

2. PROBLEMS OF SPECIALIZING OBJECTS

This section presents a few examples in which naive specialization in terms of object manipulation could generate wrong residual programs.

2.1 Object Creation

The first example shows the problems of over-specialization of object creation. The following class defines a one-dimensional point. The method `make` creates a point, modifies its state, and then returns it:

```
1  class Point {
2    int x = 0;
3    void update (int a) { x = x + a; }
4    static Point make (int s, int d) {
5      Point p = new Point ();
6      p.update (s);
7      p.update (d);
8      p.update (s);
9      return p;
10   }
11 }
```

Assume that we specialize the `make` method with respect to its first argument in an application that originally has the following lines:

```
int u = Console.getInt ();
Point a = Point.make (u, 7);
Point b = Point.make (u, 11);
int v = a.x + b.x;
int w = a == b;
```

Since the object construction at line 5 does not depend on dynamic arguments, a naive specializer may create the object during specialization and consider it static[1]. Because the receiver object and the argument are static, the specializer performs the method call at line 6. The method invocation at line 7 makes the object dynamic and lines 7 to 9 are residualized. This entails in particular residualization of accesses to the object `p`. With the aim of residualizing those accesses, the specializer records the object in a global variable `_p`: we say that the reference to the object is *lifted*. In the residual code, accesses to object `p` are realized through the variable `_p`:

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

2.1.1 Problems

Assume that `u` is indeed 42 and let us transform the above application into the following one that uses the previously specialized method:

```
Point _a = Point.make_res (7);
Point _b = Point.make_res (11);
int v = _a.x + _b.x;
int w = _a == _b;
```

Unexpectedly, results are different from those in the original application. This is because the object creation is specialized and the residual code reuses the lifted object `_p`. Because `_p` is uninitialized before the second invocation of `make_res`, the state of `_b` becomes different from the state of `b` in the original method. Also, since `_a` and `_b` share the same object, the second invocation changes `_a`'s state as well. Consequently, the value of `v` becomes incorrect. In addition, references `_a` and `_b` have the same values in the residual code whereas `a` and `b` had different values in the original method, hence the incorrect value of `w`. Here, naive specialization fails to deliver objects in their expected states and with their expected identities.

A conservative solution would make creation of object `p` at line 5 dynamic, so as to residualize all the operations on `p`. However, it fails to specialize the computation in the invocation of `update` at line 6.

2.2 Field Accesses

The second example illustrates problems of over-specialization of field accesses. The following program fragment creates a point object, and calls `update` twice with the same parameter:

```
Point p = new Point();
p.update(s);
p.update(s);
```

Since the method calls have the same receiver object and the same parameter, one could imagine that we can specialize the method with respect to the receiver object and the parameter and transform the program fragment as follows, where `update_gen` is a method that specializes `update` with respect to `p` and `s`:

```
Point p = new Point ();
update_res = update_gen (p, s);
update_res ();
update_res ();
```

The specializer `update_gen` would perform all the static operations, namely execution of `f` and assignment to `x`, and return a residual method. (For readability, we present the residual method as a first class function.) The residual method, as a result, is an empty method.

2.2.1 Problems

The execution of the transformed program fragment results in an object `p` in a different state because it performs the update operation only once, at specialization-time. This suggests that the specializer process and the user program that uses the specializer and the residual code should follow some rules.

First, even if a method is called with the same object as its parameter, it may not be regarded as static if the state of the object changes. In the above case, the state of the receiver object `p` should not be used for specialization, since `p`'s field changes.

Next, the specializer should not perform side-effects visible to the user program. In the transformed program above, `update_gen` changes `p`'s state. As a result, the heap against which the first invocation of `update_res` is performed is different from the heap against which `update` is first executed in the original program

Finally, even though it is not the case in our present example, there may be arbitrary statements between the execution of the specializer and the execution of the residual code. Through such statements, the running application may modify the static arguments used to produce the residual code, the latter being invalidated by those inconsistencies. We elaborate those rules in Sect. 4.

[1]We here assume simple binding-times for objects where an object has a type either fully static or fully dynamic.

3. RUN-TIME SPECIALIZATION IN BCS

Before discussing correctness of specialization, this section gives an overview of run-time specialization. Although our system, called ByteCode Specialization (BCS), is implemented for the Java Virtual Machine language (JVML), most techniques are basically common to other run-time specialization systems. The details of the BCS system are explained by the second and the fourth authors [14].

As an example, we use a method in a ray tracer that computes an intersection between a ray (from an observer to a point on a screen) with a scene, which is a collection of visual objects:

```
// set scene of objects and observer
for each point on the screen {
  // set the ray to the point
  Inter inter = ray.closestInter (observer, scene);
  // compute the color
}
```

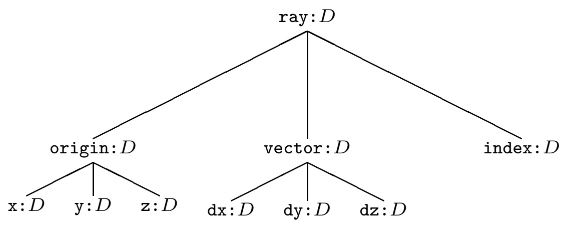
Since the position of the observer and the layout of the objects in a scene is fixed for one picture, we can specialize `closestInter` with respect to `observer` and `scene`.

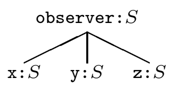
A run-time specialization system consists of three stages, namely off-line, specialization, and execution. Below, we explain the notion of binding-time specification followed by each of the three stages.

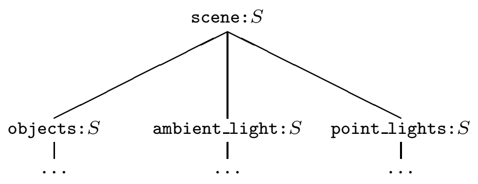
3.1 Binding-time Specification

The first step is to distinguish between static and dynamic arguments. This is the role of the *binding-time specification*. There are many ways to express binding-time specifications. The simplest solution consists in treating all the objects of a given class as either fully static or fully dynamic. Because of the emphasis on the structure of data in object-oriented languages, we opt for a more precise solution: we treat objects individually and allow them to be either fully dynamic or *partially static* (the fields of a partially static object can be either static or dynamic).

The underlying idea of our representation is that objects can be represented as trees. To each argument of the subject method, we associate one tree of binding-times: the tree is rooted at the method's argument, if the argument is an object, each field of reference type becomes a node and each field of primitive type becomes a leaf. Roots, nodes, and leaves are individually assigned a binding-time (S for static or D for dynamic) with the convention that subtrees rooted at a dynamic node are fully dynamic. In the case of our ray-tracing example, the subject method `closestInter` has three arguments (`ray`, `observer`, `scene`) and this leads to the following binding-time specification:







We simplify the notation by adopting the additional convention that subtrees rooted at a static reference are fully static if not specified otherwise. Our binding-time specification is written more concisely:

`ray`:D `observer`:S `scene`:S

The advantage of our representation is that all the heap slots reachable from static arguments can be specified individually. It should however be noted that techniques developed in this paper are also applicable to other binding-time representations.

3.2 Off-line Stage

The off-line stage, given method definitions (`closestInter` in this case) to be specialized and a binding-time specification, generates a specializer. This stage

1. performs a *binding-time analysis* to identify dynamic expressions and statements that can not be performed at specialization-time, and then

2. generates a *specializer* (a method that returns residual code at run-time) by transforming the dynamic expressions and statements in the given definitions into code-generating instructions.

3.3 Specialization Stage

The specialization stage takes place at run-time. When the specializer `closestInter_gen` is called with the values of the static arguments (`observer` and `scene`), it returns a residual method `closestInter_res`, the specialized version of the subject method `closestInter`[2].

3.4 Execution Stage

The execution stage is the execution of the residual code. Here, the residual code `closestInter_res` is called with the value of the dynamic argument, namely `ray`.

Currently, we have to manually replace a call to the subject method (`closestInter`) with a call to the residual code (`closestInter_res`) in the application program. The application program can generate more than one residual method for different static parameters. In such a case, choosing an appropriate residual method is the user's responsibility.

In our example, we can modify the application program to use the specializer as follows:

```
// set scene of objects and observer
closestInter_res = closestInter_gen (observer, scene);
for each point on the screen {
  // set the ray to the point
  Inter inter = closestInter_res (ray);
  // compute the color
}
```

[2]Actually, BCS returns an instance of a class that implements the residual code.

4. CORRECTNESS OF RUN-TIME SPECIALIZATION FOR JVML

As a first step, we review a definition of correctness of specialization for functional languages with structural equality (i.e., equality between two structured values does not depend on their locations in memory but only on the equality between corresponding fields):

Let f and f_s be functions and s be a value. f_s is a correct result of specialization of f with respect to s if, for any value d, $f(s,d) = v$ implies $f_s(d) = v'$ and v is structurally equal to v'.

Unfortunately, the definition is not directly transposable into run-time specialization for imperative languages, including JVML. This is because the definition lacks the notion of heap. For example, structural equality of results does not capture all side-effects on the heap. Also, it can not express the preservation of object identity by the program transformation. Furthermore, there is no mention of the fact that the specialization process interacts with the heap of the running application.

Below, we first formalize the heap manipulated by the Java Virtual Machine. Next, we discuss more precisely the characteristics of specialization with respect to the heap of the running application. We then state a key relation that any correct run-time specialization for JVML must satisfy. From the relation, we clarify the reasonable requirements in terms of the heap state, the specializer, the residual code, and the application process. We claim that adequacy of run-time specialization with the correctness relation and the additional soundness requirements constitutes a sound specification of the program transformation. Finally, we point out one salient characteristic of our run-time specialization for JVML that enables reuse of heap-allocated objects to improve the efficiency of the specialization.

4.1 Notations for Heaps

A *JVML value* (a value for short) is either a primitive value or a reference to an object. Two values are equal in the sense of Java's == operator: i.e., two references are equal if they reference the same address. We represent lists of JVML values by a vector-like notation: for instance, $\overrightarrow{v}$.

An object belongs to a class and has fields of values. We denote an object in class C with fields $k_0, k_1, \ldots$ as:

$$\mathit{object}_C\{k_0 = v_0, k_1 = v_1, \ldots\},$$

where v_i is the field value of k_i.

A heap is a map from references to objects. By way of example, we write a heap with an object $\mathit{object}_A\{k_0 = 42, k_1 = \beta\}$ pointed to by the reference α and an object $\mathit{object}_B\{l_0 = \alpha\}$ pointed to by a reference β (both objects point at each other) as:

$$\begin{array}{ll}\{ & \alpha \mapsto \mathit{object}_A\{k_0 = 42, k_1 = \beta\}, \\ & \beta \mapsto \mathit{object}_B\{l_0 = \alpha\} \quad \}.\end{array}$$

Given a heap H and a reference α, we can retrieve information about the object pointed to by α by means of the following three operators:

- $\mathsf{class}(\mathit{object}_C\{\ldots\}) = C$,
- $\mathsf{fields}(C)$ is a set of fields in class C, and
- $\mathit{object}_C\{\ldots, k = v, \ldots\}.k = v$.

Our execution model is single threaded. We represent method execution as follows: given method arguments $\overrightarrow{v}$ and a heap K, $m_K(\overrightarrow{v}) = \langle v, K' \rangle$ represents the execution of the method m with actual arguments $\overrightarrow{v}$ in the heap K that returns the value v and modifies the heap such that it becomes K'.

Two values on different heaps can be equivalent if not equal. This is expressed by the following definition:

Definition 1 (Equivalence Relation over Heaps) :
Let H and H' be heaps and ϕ be a surjection from references to references. $\mathcal{R}_{H,H',\phi}$ is the smallest binary relation such that:

– for all values of primitive type v, $(v, v) \in \mathcal{R}_{H,H',\phi}$, and

– for all values of reference type α, $(\alpha, \phi(\alpha)) \in \mathcal{R}_{H,H',\phi}$ iff $\mathsf{class}(H(\alpha)) = \mathsf{class}(H'(\phi(\alpha)))$ and $\forall k \in \mathsf{fields}(\mathsf{class}(H(\alpha)))$.
$(H(\alpha).k, H'(\phi(\alpha)).k) \in \mathcal{R}_{H,H',\phi}$. □

Intuitively, when v and v' are object references, $(v, v') \in \mathcal{R}_{H,H',\phi}$ means that object graphs reachable from v in H and from v' in H' have the same shape and values, ignoring the allocated addresses of objects.

4.2 Heap-Based Specialization

One of the important characteristics of run-time specialization is that the specializer and the residual code use the same heap. This is a crucial difference from compile-time specialization, in which the specializer and the residual code never share the heap. For this reason, we formalize the specialization process as a heap transformation:

$$\langle m, \overrightarrow{s}, H\rangle \xrightarrow{spec} \langle m\_res, H'\rangle,$$

where m is the subject method, $\overrightarrow{s}$ are the static arguments, H is the heap at specialization-time, m_res is the residual code, and H' is the heap after the specialization.

4.3 Correctness of Run-time Specialization

In this section, we state the correctness relation that any run-time specializer for an imperative language such as JVML must satisfy. Intuitively, this relation expresses the fact that both the subject method and the residual code, whenever run against the same arguments, yield the same result.

By "arguments," we mean not only method parameters but also reachable objects. More precisely, "same arguments" signifies that (1) parameters are equal, (2) heaps against which methods are executed are equivalent, and (3) static objects in those heaps are allocated at the same addresses. Equivalence of heaps accounts for structural equality of reachable objects. The fact that static objects must be allocated at the same addresses is a consequence of reference lifting. Indeed, static references can be lifted during specialization and residualized in such a way that execution of the residual code may access (static) objects without any need for a (static) reference to be passed to it. So static objects cannot be placed elsewhere without compromising correctness, even if structural equivalence is preserved.

We call *static heap* of a specialization that part of the heap at specialization-time that consists of objects pointed to by static references. The fact that static objects are not moved around is expressed by the following definition:

Definition 2 (Static Heap Preservation) :
Let m and m_res be methods, H and H' be heaps, and $\overrightarrow{s}$ be a list of values such that $\langle m, \overrightarrow{s}, H\rangle \xrightarrow{spec} \langle m\_res, H'\rangle$.

Let K be a heap. K preserves the static heap of the specialization iff $\forall v \in \overrightarrow{s}.(v, v) \in \mathcal{R}_{H,K,id}$, where id is the identity function. □

By "same result," we mean that (1) the return values are equal primitive values or equivalent objects and that (2) the resulting heaps (i.e., the set of live objects after the execution) are equivalent. Note that we take the latter into account as a part of the results of the method execution. This is because JVML methods can modify the state of objects that may not be reachable from the return value.

The following definition expresses the correctness relation that any run-time specialization for JVML must satisfy:

Definition 3 (Correct Specialization) :
Let m and m_res be methods, H and H' be heaps, and $\overrightarrow{s}$ be a list of values such that $\langle m, \overrightarrow{s}, H\rangle \xrightarrow{spec} \langle m\_res, H'\rangle$.

m_res is a correct result of run-time specialization of m with respect to $\overrightarrow{s}$ and H iff:

- for any list of values $\overrightarrow{d}$,
- for any heap K that preserves the static heap of the specialization, and
- for any list $\overrightarrow{l}$ of references live after the execution of m_res,

if $m_K(\overrightarrow{sd}) = \langle v, J\rangle$, then $m\_res_K(\overrightarrow{d}) = \langle v', J'\rangle$ and there exists a surjection ϕ from references to references such that $(v, v') \in \mathcal{R}_{J,J',\phi}$ and $\forall w \in \overrightarrow{l}.(w, w) \in \mathcal{R}_{J,J',\phi}$. □

4.4 Soundness Requirements

In this section, we discuss soundness requirements to ensure correct run-time specialization.

The specializer, the residual code and the application that uses the specializer and the residual code share the same heap. As a consequence, run-time specialization can be easily misused, as we observe in the example of Sect. 2.2: the specializer and the running application may modify the static heap in such a way that the residual code may be invalidated, and the specializer may modify the static heap in such a way that the heap of the running application may be made inconsistent.

In practice, there are several ways to deal with those problems. Some solutions are simple but lead to poor specialization (for instance, one can prohibit side-effects during specialization). Better specialization may be achieved thanks to more involved techniques (for instance, all side-effects may be allowed as long as a mechanism for recovering a consistent heap after the specialization is provided).

Below, we expose the solutions we adopt for BCS. They are elaborated enough to enable production of efficient code as shown in Sect. 4.5, yet simple enough to be easily implemented as shown in Sect. 5.

Our first soundness requirement is to prohibit modification of the static heap by the running application. In practice, we request the application programmer to use the residual code without modifying the objects reachable from static parameters.

Our second soundness requirement is to prohibit potentially offending side-effects during specialization. Not all side-effects are potentially offending: only those whose effect is visible from the outside of the specialization process need to be forbidden. In other words, we request that the specialization process preserves the heap of the running application:

Definition 4 (Heap Preserving Specialization) :
Let m and m_res be methods, H and H' be heaps, and $\overrightarrow{s}$ be a list of values such that $\langle m, \overrightarrow{s}, H\rangle \xrightarrow{spec} \langle m\_res, H'\rangle$.

Let $\overrightarrow{l}$ be a list of live references after this specialization. The specialization preserves the heap of the running application iff $\forall v \in \overrightarrow{l}$, $(v, v) \in \mathcal{R}_{H,H',id}$. □

As long as this condition is satisfied, specialization processes can statically perform side-effects to possibly yield more efficient residual code (see Sect. 5.3).

Those two soundness requirements ensure in particular that the heap against which the residual code is executed always preserves the static heap (Definition 2). As a result, the residual code can directly access to any objects reachable from static parameters at its run-time.

4.5 Specialization Store Reuse

We observe in the example of Sect. 2.1 that we can produce more efficient residual code by performing dynamic allocation of some objects at specialization-time and reuse them in the residual code. However, we also show that naive reuse of such objects may also break the semantics of the running application because of the loss of objects' identity and state. The issue is subtle since problems can only be identified when the residual code is used several times.

Regarding in particular the preservation of object identity, one expects that the residual code always returns the same reference only if the subject method does so:

Property 1 (Unique Identity Generation) :
Let m and m_res be methods, H and H' be heaps, and $\overrightarrow{s}$ be a list of values such that $\langle m, \overrightarrow{s}, H\rangle \xrightarrow{spec} \langle m\_res, H'\rangle$.

Specialization satisfies the 'unique identity generation' property iff, for any application P, for any two executions of m:

$$m_{K_1}(\overrightarrow{s\ d_1}) = \langle v_1, K'_1\rangle \text{ and } m_{K_2}(\overrightarrow{s\ d_2}) = \langle v_2, K'_2\rangle,$$

if we substitute m for m_res:

$$m\_res_{K_1}(\overrightarrow{d_1}) = \langle v_1^{res}, {K'}_1^{res}\rangle \text{ and } m\_res_{K_2}(\overrightarrow{d_2}) = \langle v_2^{res}, {K'}_2^{res}\rangle,$$

then for any lists $\overrightarrow{l_1}$ and $\overrightarrow{l_2}$ of references live after, respectively, the first and the second execution of m_res, there exists two surjections ϕ_1 and ϕ_2 from references to references such that:

$$(v_1, v_1^{res}) \in \mathcal{R}_{K'_1,{K'}_1^{res},\phi_1} \text{ and } \forall w \in \overrightarrow{l_1}.(w, w) \in \mathcal{R}_{K'_1,{K'}_1^{res},\phi_1},$$

and

$$(v_2, v_2^{res}) \in \mathcal{R}_{K'_2,{K'}_2^{res},\phi_2} \text{ and } \forall w \in \overrightarrow{l_2}.(w, w) \in \mathcal{R}_{K'_2,{K'}_2^{res},\phi_2},$$

and

$$\forall x \in \{v_1\}\cup\overrightarrow{l_1}.\forall y \in \{v_2\}\cup\overrightarrow{l_2}.\text{ if } \phi_1(x) = \phi_2(y) \text{ then } x = y. \ \square$$

One approach to guarantee that this property is satisfied is the conservative, systematic residualization of the construction of dynamically allocated objects. It is however not obvious how to preserve this correctness property while enabling reuse of heap-allocated objects as pointed out in Sect. 2.1.

The correctness definitions and the soundness requirements however allow for an adequate optimization: *specialization store reuse*. Concretely, BCS distinguishes operations that create and/or modify objects in terms of the visibility of the objects from the computation that follows the subject method invocation:

- for operations for *may-visible* objects (i.e., objects that may be visible from the rest of the computation), BCS performs *and* residualizes those operations during specialization, and
- for operations for *never-visible* objects, BCS *only* performs those operations during specialization; the residual code merely reuses the object created during specialization.

This is an optimization because it avoids, whenever safely possible, residualization of object creation and performs more field assignments during specialization (see Sect. 5.4). It should be observed that sharing of objects among different instances of the same residual code may compromise correctness of multi-threaded programs. Although simple provisions can be made to bypass this problem, we stick to single threaded programs for the sake of simplicity.

5. IMPLEMENTATION ISSUES

This section focuses on implementation techniques that guarantee correctness of run-time specialization. The technical intricacies of the implementation for JVML are presented in the first author's paper [1]. The program transformation in itself is an extension of previous work by the second and last authors [14]. Below, we first elaborate the notions introduced in the previous section, such as side-effects visible from the outside and visible objects from the rest of the computation, from the viewpoint of implementation.

5.1 Preliminary Analyses

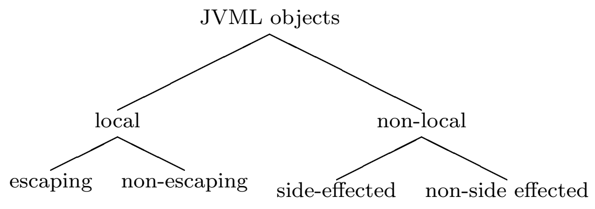
Objects manipulated during the execution of a method m can be divided into two categories:

- a *local object* to the execution of m is an object that is created during the execution of m, and
- a *non-local object* to the execution of m is an object that existed prior to the execution of m.

Local objects can further be distinguished by whether or not they are *escaping* from the execution of m, and non-local objects can be further distinguished by whether or not they are side-effected by a *visible side-effect* during the execution of m:

- a visible side-effect is a destructive update of a field of a non-local object occurring during the execution of m, and
- a local object escapes when a reference to the object become reachable from the return value of the execution of m, or from a reference live after the execution of m.

This leads to the following classification of JVML objects:



We suppose that we are given preliminary compile-time analyses that annotate the instructions of the subject method m:

- all the assignments that may have visible side-effects in some execution of m are annotated with `SE` (conversely, unannotated assignments never cause visible side-effects in any execution), and
- all the `new` instructions that may create an escaping object in some execution of m are annotated with `ESC` (conversely, unannotated `new` instructions never create an escaping object in any execution).

We now see how this information is used to guarantee correctness and soundness of run-time specialization in BCS.

5.2 Notations for Specializers and Residual Codes

Although run-time specialization in BCS is done at the bytecode level, we hereafter discuss specializers and residual code in a pseudo-language resembling Java. We informally present specializers with the following code-generating instructions:

- `GEN` *stat*, where *stat* is a Java statement or a label, emits its argument into the residual code,
- `LIFT` *expr*, where *expr* is a Java expression, evaluates its argument and emits a constant expression that yields the evaluated value of *expr*, and
- `INL` *m_gen* $(a_0, a_1, \dots)$, where *m_gen* is a specializer and a_i its arguments, inlines the code resulting from the specialization *m_gen* $(a_0, a_1, \dots)$ and possibly returns a (static or dynamic) value.

5.3 Heap Preserving Specialization

In order to preserve static heaps during specialization, as required in Definition 4, we prohibit visible side-effecting assignments by adding binding-time analysis typing rules to make those assignments dynamic.

Consider the example of Sect. 2.2, with annotations of visible side-effects:

```
class Point {
  int x = 0;
  void update (int a) { x =SE x + a; }
}
```

We specialize the `update` method with respect to the enclosing point[3] and the argument `a`. The destructive update of the coordinate is a visible side-effect, it is therefore made dynamic by the binding-time analysis. The right-hand side of the assignment is a function call whose arguments are static, it is therefore evaluated during specialization.

We show in Fig. 1 the specializer and the residual code when the coordinate of the enclosing point is 0 and the argument `a` is 42.

[3]We regard the receiver object of a method invocation as a parameter.

```
update_gen (Point p, int a) {
  GEN p.x = LIFT (x + a);
}

void update_res () {
  p.x = 42;
}
```

Figure 1: Specializer and residual code for `update`

5.4 Unique Identity Generation

To preserve the identity of objects, as specified by Property 1, we enforce residualization of the construction of escaping objects.

Consider the example of Sect. 2.1, with appropriate escaping information:

```
class Point {
  int x = 0;
  void update (int a) { x = x + a; }
  static Point make (int s, int d) {
    Point p = new^ESC Point ();
    p.update (s);
    p.update (d);
    p.update (s);
    return p;
  }
}
```

We specialize `make` with respect to the argument `s`. During specialization, the construction of the escaping object is both residualized and performed and the created object is considered static. The next three method calls are all inlined. In the case of the first one, the receiver object and the argument are static; the side-effect is both residualized and performed because it is visible and the modified object is escaping. In the case of the next method call, the side-effect is residualized because the argument is dynamic. As a result, the state of the receiver object becomes dynamic and the last side-effect must also be residualized.

The specializer and the residual code resulting from `s` being given the value 42 are shown in Fig. 2.

```
make_gen (int s) {
  Point p = new Point ();
  GEN Point _p = new Point ();
  INL (update_gen_0 (p, s));
  INL (update_gen_1 (p, d););
  INL (update_gen_1 (p, s));
  GEN return _p;
}

static Point make_res (int d) {
  Point _p = new Point ();
  _p.x = 42;
  _p.x = 42 + d;
  _p.x = _p.x + 42;
  return _p;
}
```

Figure 2: Specializer and residual code for `make`

Contrary to the naive specialization of the same sample code in Sect. 2.1, the ‘unique identity generation’ property is here satisfied. Additionally, we observe that the specializer safely performs some side-effects and yields more efficient residual code, since one update is evaluated away. In the next section, we discuss more evidence that BCS performs better specialization for object-oriented programs.

6. OBJECT-ORIENTED OPTIMIZATIONS

Ubiquitous object creation and virtual dispatching are typical overheads of object-oriented programs. The following examples illustrate how BCS targets those inefficiencies.

6.1 Object Construction Elimination

To relieve the residual code from useless dynamic allocations, specialization can evaluate away creation of non-escaping objects.

The class below represents complex numbers and the `eval` method performs simple arithmetic operations:

```
class Complex {
  float real, imag;
  Complex times (Complex d) {
    Complex mul = new^ESC Complex ();
    mul.real = real * d.real − imag * d.imag;
    mul.imag = real * d.imag + imag * d.real;
    return mul;
  }
  Complex plus (Complex s) {
    Complex add = new Complex ();
    add.real = real + s.real;
    add.imag = imag + s.imag;
    return add;
  }
  static Complex eval (Complex s, Complex d) {
    return s.plus(s).times(d);
  }
}
```

We specialize `eval` with respect to the argument `s`, whose value is $object_{\texttt{Complex}}\{real = 2, imag = 3\}$. The corresponding specializer `eval_gen` is shown in Fig. 3.

```
1   times_gen (Complex t) {
2     Complex mul = new Complex ();
3     GEN Complex mul = new Complex ();
4     GEN mul.real =
5         LIFT(t.real)*_d.real − LIFT(t.imag)*_d.imag;
6     GEN mul.imag =
7         LIFT(t.real)*_d.imag + LIFT(t.imag)*_d.real;
8     return mul;
9   }
10  plus_gen (Complex s, Complex s) {
11    Complex add = new Complex ();
12    add.real = s.real + s.real;
13    add.imag = s.imag + s.imag;
14    return add;
15  }
16  eval_gen (Complex s) {
17    GEN return INL times_gen (INL plus_gen (s, s), d);
18  }
```

Figure 3: Specializer for `eval`

At the beginning of specialization, the method `plus_gen` is called. Line 11 creates a static object that we call α hereafter. Lines 12 and 13 initialize the fields of α which represents the complex value $4 + 6i$ when it is returned. Note that no residual code is generated during the execution of the method `plus_gen`. The method `times_gen` is then called. Line 2 creates a static object that we call β. Object creation is also residualized because it is originally annotated with `ESC`. Accesses to the `real` and `imag` fields of α in the statements at lines 4 and 6 are performed at specialization-time but the rest of the arithmetic operations must be residualized. Because the creation site of β is annotated with ESC, field assignments are both performed and residualized. The resulting residual code is shown in Fig. 4.

```
static Complex eval_res (Complex _d) {
  Complex mul = new Complex ();
  mul.real = 4 * _d.real − 6 * _d.imag;
  mul.imag = 4 * _d.imag + 6 * _d.real;
  return mul;
}
```

Figure 4: Residual code for `eval`

The results of the escape analysis are used to avoid residualizing creation of the non-escaping object that originally occurred inside the `plus` method. Whereas the execution of the subject method creates one non-escaping local object as an intermediate result of the computation, we have been able, by using the results of the escape analysis, to remove any need for it in the residual code.

6.2 Specialization Store Reuse

The following example, which is taken from a study on compile-time partial evaluation of imperative languages [2], demonstrates specialization of the construction of a data structure.

The method below dynamically allocates a list, initializes it, and then looks up a key element:

```
public int search (int n, int data, int key) {
  // make a list
  List list = new List ();
  list.key = 0;
  list.data = data++;
  List ptr = list;
  for (int i = 1; i < n; i++) {
    ptr.next = new List ();
    ptr = ptr.next;
    ptr.key = i;
    ptr.data = data++;
  }
  // look up the key element
  for (ptr = list; ptr.key != key; ptr = ptr.next)
    ;
  return ptr.data;
}
```

Let us assume that `n` and `key` are static. The corresponding specializer is shown in Fig. 5. Line 2 creates an object, which we call α_0 hereafter. Line 3 sets `key` field of α_0 to 0, while line 4 is residualized because `data` is dynamic. The loop from line 6 to line 11 is static. All the constructions at line 7 and all the assignments at lines 8 and 9 are done during specialization, resulting in a series of partially initialized objects $\alpha_1, \alpha_2, \ldots, \alpha_{n-1}$. The loop statement from line 12 to 13 is also static, resulting on `ptr` being eventually initialized to $\alpha_{\texttt{key}}$. Eventually, the return statement is residualized, the `ptr` variable lifted, and field access residualized because of the contents of `data` being dynamic. The corresponding residual code is shown in Fig. 6.

During specialization, all the object creations are performed and remembered. When executed, the residual code does not create any object on its own but instead reuses those objects. Since execution of the residual code modifies the state of the objects, one may fear that multiple executions of the residual code may create inconsistencies. This

```
1  search_gen (int n, int key) {
2    List list = new List ();
3    list.key = 0;
4    GEN list.data = data++;
5    List ptr = list;
6    for (int i = 1; i < n; i++) {
7      ptr.next = new List ();
8      ptr = ptr.next;
9      ptr.key = i;
10     GEN ptr.data = data++;
11   }
12   for (ptr = list; ptr.key != key; ptr = ptr.next)
13     ;
14   GEN return ptr.data;
15 }
```

Figure 5: Specializer for `search`

```
public int search_res (int data) {
  α_0.data = data++;
  α_1.data = data++;
  α_2.data = data++;
  ...
  α_{n−1}.data = data++;
  return α_key.data;
}
```

Figure 6: Residual code for `search`

actually does not occur because specialization has residualized all the assignments that initialize them (here, the `data` fields' increments).

This reuse of already allocated objects is actually reminiscent of an optimization technique used in object-oriented languages with automatic memory management which consists in recycling memory slots to avoid ubiquitous object creation.

6.3 Virtual Dispatching

Like optimization based on compile-time class-hierarchy analysis, run-time specialization in BCS can eliminate virtual dispatches. It is however more efficient because (1) the specializer knows the run-time class of static objects and (2) optimizations provided by specialization (such as loop unrolling) disambiguate method calls.

We show below the source code of the subject method `closestInter` from the ray tracer discussed in Sect. 3:

```
class Ray {
  public Inter closestInter (Point observer, Scene scene) {
    float smallest = Float.MAX_VALUE;
    Inter closest_inter = null;
    Inter tmp = null;
    for (int i = 0; i < scene.objects.length;) {
      tmp = scene.objects[i].intersect (observer, this);
      i++;
      if (tmp != null) {
        if (tmp.k < smallest) {
          smallest = tmp.k;
          closest_inter = tmp;
        }
      }
    }
    return closest_inter;
  }
}
```

The scene to be displayed is composed of objects that are gathered together in the array `scene.objects` that appears at line 7. The elements of `scene.objects` have for declared class an abstract class whose subclasses implement the different kinds of objects that may appear in the scene (`Plane`, `Sphere`, ...). Since each subclass has its own implementation of the `intersect` method, it is only when execution reaches line 7 that we discover where it pursues. Residualization of the method call is a systematic solution to this ambiguity but it fails to produce optimized residual code in general. Indeed, if the receiver object is static, knowledge of its run-time class during specialization enables residualization of dedicated residual code. BCS performs this optimization.

We show below the specializer and the residual code of the specialization of method `closestInter` with respect to the `scene` and `observer` arguments. Specialization of the `closestInter` method entails specialization of all the involved implementations of `intersect`. In the specializer (Fig. 7), `intersect_gen` is a wrapper that is in charge of calling the adequate specializer (depending on whether the receiver object is a plane, a sphere, ...). The residual code resulting from the specialization of `intersect` methods is inlined in the residual code (not displayed here for the sake of clarity) (Fig. 8).

```
closestInter_gen (Observer observer, Scene scene) {
  float smallest = Float.MAX_VALUE;
  Inter closest_inter = null;
  Inter tmp = null;
  GEN float _smallest = LIFT (smallest);
  GEN Inter _closest_inter;
  GEN Inter _tmp;
  for (int i = 0; i < scene.objects.length;) {
    GEN _tmp = INL
        (intersect_gen (scene.objects[i], observer));
    i++;
    GEN if (_tmp.k < _smallest) goto L(i)
    GEN _smallest = _tmp.k;
    GEN _closest_inter = _tmp;
    GEN L(i):
  }
  GEN return _closest_inter;
}
```

Figure 7: Specializer for `closestInter`

```
Inter closestInter_res (Ray ray) {
  float _smallest = Float.MAX_VALUE;
  Inter _closest_inter;
  Inter _tmp;
  _tmp = /* inlined code that computes the
             intersection  with the  first  object */;
  if (_tmp.k < _smallest) goto L0
  _smallest = _tmp.k;
  _closest_inter = _tmp;
  L0:
  _tmp = /* inlined code that computes the
             intersection  with the second object */;
  if (_tmp.k < _smallest) goto L1
  _smallest = _tmp.k;
  _closest_inter = _tmp;
  L1:
   // ...
  return _closest_inter;
}
```

Figure 8: Residual code for `closestInter`

7. PERFORMANCE MEASUREMENTS

We have implemented the system described above by extending the second and the last author's system [14]. We have chosen three Java applications that are written in an object-oriented style to measure performance.

We compare for each application the execution time of a bottleneck, generic method with the execution of its corresponding specialized version, generated by BCS. We also measure the overhead of Just-in-time compilation in each case. Concretely, we run the same methods with the same arguments 1,000,000 times and took the average execution time at each 10,000 runs slice. To estimate the execution time of an iteration and of the Just-in-time overhead, we do a linear approximation by linear regression in the least square sense of the second half of the data, i.e., after the Just-in-time overhead.

The experiments are carried in three different environments: (1) virtual machine: Sun JDK 1.3.1 Hotspot Client; architecture: UltraSparc II at 400MHz (Sun Ultra Enterprise 4500, 14 processors); operating system: Solaris 2.8, (2) virtual machine: Sun JDK 1.3.1 Hotspot Client; architecture: Pentium III at 700MHz (IBM Netfinity 7100, 4 processors); operating system: Linux 2.4.6. (3) virtual machine: IBM JDK 1.3.0 Classic (JIT enabled); architecture: Pentium III at 700MHz (IBM Netfinity 7100, 4 processors); operating system: Linux 2.4.6.

Our first example is an implementation proposed by Schultz [18] of the power function written using the Strategy design pattern. The function is specialized with respect to its exponent of value 19, the multiplication operator and its neutral; the raised base value is dynamic (results in Table 1).

env.		unspec. (μs)	spec. (μs)	speed-up
(1)	method	1.99	1.37	1.46
	JIT	40,039	181,237	
(2)	method	0.54	0.42	1.30
	JIT	12,849	27,392	
(3)	method	0.32	0.07	4.35
	JIT	48,921	54,081	

Table 1: Power Function

Our second example is an application that displays Mandelbrot sets [14]. The formula to be displayed is input interactively and is evaluated by an interpreter written in an object-oriented way. We specialize the evaluation method of that interpreter with respect to the expression $z * z + c$ (results in Table 2).

env.		unspec. (μs)	spec. (μs)	speed-up
(1)	method	2.08	1.93	1.07
	JIT	151,695	103,046	
(2)	method	1.14	1.20	0.95
	JIT	30,553	34,650	
(3)	method	1.49	1.42	1.05
	JIT	109,252	123,235	

Table 2: Mandelbrot Sets

Our third example is a ray tracer implemented in Java by using a standard text book on the model of experiments for another (compile-time) specialization system [3]. This is the same experiment as the one described in Sect. 3: the method `closestInter` is here specialized with respect to an observer at a fixed position and a scene of ten objects (results in Table 3).

env.		unspec. (μs)	spec. (μs)	speed-up
(1)	method	10.18	8.65	1.18
	JIT	196,055	200,485	
(2)	method	6.40	5.12	1.25
	JIT	115,241	104,031	
(3)	method	9.87	7.84	1.26
	JIT	208,341	557,194	

Table 3: Ray Tracer

We have done the power function experiment for the purpose of comparison with the JSpec compile-time specializer for Java [18]. Results obtained with IBM's virtual machine under the Intel architecture are actually comparable while results with Sun's Hotspot virtual machine are below our expectations.

The optimization that run-time specialization in BCS provides in the case of the Mandelbrot sets drawer essentially amounts to devirtualization of method calls. The displayed formula is in fact very simple and does not involve many virtual dispatches. As a consequence, performance gains are not substantial.

Run-time specialization in BCS also features traditional optimizations such as constant propagation, expression unfolding, and loop unrolling. The ray-tracing experiment benefits from both devirtualization and traditional optimizations, and this leads to better speed-up. Yet, similar experiments with other specialization systems (compile-time specialization [3] and run-time specialization [7]) reach even better speed-up. We believe that this is due (1) to less aggressive compilation optimizations in Just-in-time compilers compared to traditional compilers, (2) to the lack of optimizations in BCS during the off-line stage, and (3) to the fact that some optimizations performed by Just-in-time compilers and specialization are redundant.

The combined use of dynamic compilation (in the form of JIT compilation) and run-time specialization (that often entails code growth) complicates performance measurements. Indeed, JIT compilation already has an overhead and one must be careful that specialization does not make it much worse. Although figures are reassuring, their interpretation is difficult because we are not provided with reliable descriptions of the behavior of JIT compilation. Ideally, JIT compilation and run-time specialization should be studied in a common framework to gain more confidence about the concurrent action of optimizations and overheads.

Above experiments do not benefit from object construction elimination and specialization store reuse. This is because the current implementation appeals to the Reflection API of the Java platform and object traversals to residualize initialization of escaping objects. Those techniques lack efficiency in themselves. They impose a consequent specialization overhead and limit the range of real-size examples that we can investigate. We are now in the process of designing a more efficient approach whose implementation will let us show stronger evidence of performance gains.

8. RELATED WORK

C-Mix [2] is a compile-time specializer for the C language. It deals with pointers and structures, which are in essence much like Java references and objects. There are other similarities. For instance, replacement of dynamic allocation with static allocation by C-Mix can be compared to our specialization store reuse optimization. However, run-time specialization in BCS benefits from the fact that references are liftable and is simpler in many respects.

Tempo [5] provides both compile-time and run-time specialization for the C language. Formal presentation of the run-time specialization facility does not handle dynamic allocation nor structures and run-time specializers are fairly different since they simply copy precompiled templates to memory, thus preventing run-time optimizations. Yet, the amortization cost is very small.

Lea and Dean et al. first discussed specialization of object-oriented languages but focused on virtual dispatching [6], [11], not on partial evaluation techniques.

JSpec [19] is a compile-time specializer for the Java language. The specialization is off-line and uses Tempo as a compile-time specializer for C and translators between the Java and the C languages. Objects are supported through *specialization classes* [21]. The formalization by Schultz [17] does not handle mutable objects nor dynamic allocation and focuses instead on interactions between objects.

Fujinami [7] proposes a run-time specializer for the C++ language. Specialization is done on objects' methods with respect to the fields of the enclosing object and the residual code can be used until invalidation of the contents of these fields. The specializer produced at compile-time goes through an optimization layer, whereas our implementation relies exclusively on the optimizations provided by Just-in-time compilers.

Thiemann proposes a region-based binding-time analysis [20]. It uses effects to determine binding-times and it makes possible to specialize dynamic allocation and accesses to data structures.

Asai integrates partial evaluation into a language interpreter [4] and discusses specialization based on the heap state at the time of specialization. Our work can be seen as a similar program transformation for non-interactive use. An important difference is that we do not assume side-effect free subject methods.

9. CONCLUSION AND FUTURE WORK

In this paper, we presented a run-time specialization system for the Java language that soundly and effectively specializes programs with references and objects.

Our contributions include a definition of sound run-time specialization in presence of heap-allocated objects. Regarding side-effects and dynamic allocation, we propose non-conservative solutions, with in particular a mechanism for reusing objects created during specialization. We also explain how we implement our specialization strategy and show how it achieves production of efficient residual code for object-oriented programs. Our implementation for Java is an extension of previous work by the second and last authors [14] and our experiments show, for instance, approximately 20-25% speed-up factor for a ray-tracing application.

Now that we have explained in details our specialization strategy, we are interested in the accompanying semantics, with the aim of effectively proving correctness, including in particular correctness of the specialization store reuse optimization. We also think that alternative approaches for sound run-time specialization, concerning for instance visible side-effects during specialization, deserve investigation. In order to complete our implementation, we plan to use third parties compile-time analyses to annotate the subject program with information about side-effects and escaping. A properly enhanced implementation will let us study larger examples in order to attest the relevance of our design choices. Ultimately, we would like to support the whole JVML language. In this perspective, pertinent solutions to handle exceptions, subroutines and others will be supported.

10. ACKNOWLEDGMENTS

The authors are grateful to Kenichi Asai for his help and would like to thank anonymous reviewers and the members of Prof. Yonezawa's group at the University of Tokyo for their comments.

11. REFERENCES

[1] Affeldt, R. 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] Andersen, L. O. Binding-time analysis and the taming of C pointers. In *Partial Evaluation and Semantics-Based Program Manipulation* (Jun. 1993), ACM Press, pp. 47–58.

[3] Andersen, P. H. Partial evaluation applied to ray tracing. In *Software Engineering in Scientific Computing* (1996), W. Mackens and S. Rump, Eds., Vieweg, pp. 78–85.

[4] Asai, K. Integrating partial evaluators into interpreters. In *Second International Workshop on Semantics, Applications and Implementation of Program Generation* (Sep. 2001), vol. 2196 of *Lecture Notes in Computer Science*, Springer-Verlag.

[5] Consel, C., and Noël, F. A general approach for run-time specialization and its application to C. In *23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Jan. 1996), ACM Press, pp. 145–156.

[6] Dean, J., Chambers, C., and Grove, D. Selective specialization for object-oriented languages. *ACM SIGPLAN Notices 30*, 6 (1995), 93–102.

[7] Fujinami, N. Automatic run-time code generation in C++. In *Scientific Computing in Object-Oriented Parallel Environments* (1997), vol. 1343 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 9–16.

[8] Fujinami, N. Determination of dynamic method dispatches using run-time code generation. In *Types in Compilation* (Mar. 1998), X. Leroy and A. Ohori, Eds., vol. 1473 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 253–271.

[9] Grant, B., Mock, M., Philipose, M., Chambers, C., and Eggers, S. J. Annotation-directed run-time specialization in C. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (Jun. 1997), ACM Press, pp. 163–178.

[10] Grant, B., Philipose, M., Mock, M., Eggers, S. J., and Chambers, C. An evaluation of staged run-time optimizations in DyC. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (May 1999), ACM Press.

[11] Lea, D. Customization in C++. In *C++ Conference* (1990), pp. 301–314.

[12] Lee, P., and Leone, M. Optimizing ML with run-time code generation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (1996), ACM Press, pp. 137–148.

[13] Leone, M., and Lee, P. Lightweight run-time code generation. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (Jun. 1994), ACM Press, pp. 97–106.

[14] Masuhara, H., and Yonezawa, A. A portable approach to dynamic optimization in run-time specialization. *Journal of New Generation Computing 20*, 1 (Nov. 2001), 101–124. An extended version of [15].

[15] Masuhara, H., and Yonezawa, A. Run-time bytecode specialization: A portable approach to generating optimized specialized code. In *Programs as Data Objects* (2001), O. Danvy and A. Filinski, Eds., vol. 2053 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 138–152.

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

[17] Schultz, U. Partial evaluation for class-based object-oriented languages. In *Programs as Data Objects* (2001), O. Danvy and A. Filinski, Eds., vol. 2053 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 173–197.

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

[19] Schultz, U. P., Lawall, J. L., Consel, C., and Muller, G. Toward automatic specialization of Java programs. In *13th European Conference on Object-Oriented Programming* (Jun. 1999).

[20] Thiemann, P. J. Correctness of a region-based binding-time analysis. In *Mathematical Foundations of Programming Semantics, Thirteenth Annual Conference* (1997), vol. 6 of *Electronic Notes in Theoretical Computer Science*, Elsevier Science. Available at `http://www.elsevier.nl/locate/entcs/volume6.html`.

[21] Volanschi, E. N., Consel, C., Muller, G., and Cowan, C. Declarative specialization of object-oriented programs. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (1997), ACM Press.