A Portable Approach to Dynamic Optimization in Run-time Specialization

Hidehiko Masuhara

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

masuhara@acm.org

Akinori Yonezawa

Department of Information Science, University of Tokyo

yonezawa@is.s.u-tokyo.ac.jp

Received 18 June 2001. Revised 28 July 2001

Abstract This paper proposes a *run-time bytecode specialization* (BCS) technique that analyzes programs and generates specialized programs at run-time in an intermediate language. By using an intermediate language for code generation, a back-end system can *optimize the specialized programs after specialization.* The system uses Java virtual machine language (JVML) as the intermediate language, which allows the system to easily achieve practical *portability* and to use existing sophisticated just-in-time (JIT) compilers as its back-end. The binding-time analysis algorithm is based on a type system, and covers a non-object-oriented subset of JVML. The specializer generates programs on a per-instruction basis, and can perform *method inlining at run-time.* Our performance measurements show that a non-trivial application program specialized at

* To appear in *Journal of New Generation Computing*, 20(1), Nov. 2001.
This paper is an extended version of "A Portable Approach to Generating Optimized Specialized Code", in *Proceedings of Second Symposium on Programs as Data Objects (PADO-II)*, Lecture Notes in Computer Science, vol.2053, pp.138–154, Aarhus, Denmark, May 2001[23].

run-time by BCS runs approximately 3–4 times faster than the unspecialized one. Despite the large overhead of JIT compilation of specialized code, we observed that the overall performance of the application can be improved.

Keywords Program Specialization, Partial Evaluation, Run-time Code Generation, Just-In-Time Compilation, Java Virtual Machine Language, Intermediate Language

§1 Introduction

Given a generic program and the values of some parameters, *partial evaluation* techniques generate a specialized program with respect to the values of those parameters[13, 19]. Most of those techniques have been studied as *source-to-source* transformation systems; *i.e.,* they analyze programs in a high-level language and generate specialized programs in the same language. They have been successful in the optimization of various programs, such as interpreters, scientific application programs, and graphical application programs[5, 15, 22].

Run-time specialization (RTS) techniques[12, 14, 20, 24] efficiently perform partial evaluation at run-time (1) by constructing a *specializer* (or a *generating extension*) for each source program at compile-time and (2) by directly generating native machine code at run-time. The drastically improved specialization speed enables programs to be specialized by using values that are computed at run-time, which means that RTS provides more specialization opportunities than compile-time specialization. Several studies reported that RTS can improve performance of programs for numerical computation[20, 24], an operating system kernel[27], an interpreter of a simple language[20], etc.

One of the problems of RTS systems is a trade-off between efficiency of specialization and efficiency of specialized code. For example, Tempo[24] generates specialized programs by merely copying pre-compiled native machine code. The performance of the generated code is 20% slower than that is generated by compile-time specialization on average. Of course, we could optimize specialized programs at run-time by optimizing generated code after specialization. It however makes *amortization*[*1] more difficult.

In this paper, we describe an alternative approach called *run-time bytecode*

[*1] In RTS systems, a specialization process of a procedure is *amortized* if the amount of reduced execution time of the procedure becomes larger than the time elapsed for the specialization process.

specialization (BCS), which is an automatic *bytecode-to-bytecode* transformation system. The characteristics of our approach are: (1) the system directly analyzes program and constructs specializers in a bytecode language; and (2) the specializer generates programs in the bytecode language, which makes it possible to apply optimizations *after specialization* by using just-in-time (JIT) compilation techniques.

As the bytecode language, we choose the Java virtual machine language (JVML)[21], which provides us practical portability. The system can use existing compilers as its front-end, and widely available Java virtual machines, which may include sophisticated JIT compilers, as its back-end. The analysis of JVML programs is based on a type system derived from the one for JVML[31]. A specializer can be basically constructed from the result of the analysis, and can perform method inlining at run-time.

Thus far, we have developed our prototype system for a non-object-oriented subset of JVML; the system support only primitive types, arrays, and static methods. Although the system does not yet support important language features in Java, such as objects and virtual methods, it has sufficient functionality to demonstrate fundamental costs in our approach, such as efficiency of specialized code and overheads of specialization and JIT compilation.

The rest of the paper is organized as follows. Section 2 overviews existing RTS techniques and their problems. BCS is described in Section 3. Section 4 presents the performance measurement of our current implementation. Section 5 discusses related studies. Section 6 concludes the paper.

§2 Run-Time Specialization

2.1 Program Specialization

An offline partial evaluator processes programs in two phases: *binding-time analysis* and *specialization.* The binding-time analysis phase takes a program and a list of the binding-times of arguments of a method in the program and returns an annotated program in which every sub-expression is associated with binding-time. The binding-time of an expression is *static* if the value can be computed at specialization time or *dynamic* if the value is to be computed at execution time. For example, when a program shown in Fig. 1 (a) is analyzed with list [dynamic, static] as the binding-times of `x` and `n`, an annotated program shown in Fig. 1 (b) is returned. In the annotated program, an un-

```
class Power {
 static int power(int x, int n)
 {
   if (n==0)
     return 1;
   else
     return x*power(x,n-1);
 }
}
```

(a) Original program

```
class Power {
 static int power(int x, int n)
 {
   if (n==0)
     return 1;
   else
     return x*power(x,n-1);
 }
}
```

(b) Binding-time annotated program

Fig. 1 An example program and its binding-time annotations

derlined expression denotes dynamic binding-time, and a non-underlined one denotes static.

In the specialization phase, the annotated program is executed with the values of the static parameters, and a specialized program is returned as a result. The execution rules for static expressions are the same as the ordinary ones. The rule for the dynamic expressions is to return the expression itself. For example, execution of annotated `power` with argument 3 for static parameter `n` proceeds as follows: it tests "`n==0`", then selects the 'else' branch, computes `n-1`, and recursively executes `power` with 2 (*i.e.,* the current value of `n-1`) as an argument. It eventually receives the result of the recursive call, which should be "`x*x*1`", and finally returns "`x*x*x*1`" by appending the received result to "`x*`".

2.2 Overview of Run-time Specialization

Run-time specialization (RTS) techniques efficiently specialize programs by generating specialized programs at machine code level[12, 14, 18, 20, 24, 33].

Given a source program and binding-time information, an RTS system efficiently generates specialized programs at run-time in the following way. It first performs binding-time analysis on the source program, similar to compile-time specialization systems. It then generates a program called *specializer* from a binding-time annotated program by substituting each dynamic expression with a *code generator,* which generates machine instructions of the expression when executed. At run-time, when a specializer is executed with a static input, it executes the static expressions, and directly generates a specialized program at

machine code level.

By directly generating a specialized program in a machine language, RTS systems are fast enough to be executed at run-time. They thus give more specialization opportunities; they can specialize programs with respect to values that can only be obtained at run-time.

2.3 Problems

Implementation of a code generator is one of the problems of RTS systems. A code generator should be fast and should generate efficient code the same time. Since a code generator have to generate machine instructions from a *fragment* of the original program, a special-purpose compiler is needed.

[1] Efficiency.

A code generator in an RTS system should achieve both efficiency of specialization processes and efficiency of specialized code.

A program that is specialized by an RTS system is usually slower than that specialized by a compile-time specialization system. This is because RTS systems apply less optimizations at specialization-time, such as instruction scheduling and register allocation, to the specialized code for the sake of efficient specialization. Furthermore, programs that have a number of method invocations (or function calls) would be much slower since method inlining is not performed in several RTS systems. For example, Noël, et al. showed that the run-time specialized programs have 20% overheads over the compile-time specialized ones on average, in their study on Tempo[24].

If an RTS system performed optimizations at run-time, specialized programs would become faster. In fact, there are several systems that optimizes specialized code at run-time[3, 20, 26]. However, the time spent for the optimization processes makes amortization more difficult.

Consequently, an RTS system that can flexibly balance a degree of optimization of specialized code and time for generating specialized code would be beneficial.

[2] Portability.

In order to directly generate machine code, RTS systems often depend on the target machine architecture. A typical RTS system includes a special purpose compiler from source code (usually in a high-level language) to native

machine code. This is because instructions in specialized program are derived from a *fragment* of the source code, and not from a compilation unit of traditional compilers, such as classes and methods.

Several techniques have been proposed to overcome the problem. For example, Tempo uses output of a standard C compiler for building code generators[9, 24]. `C, which is a language with dynamic code generation mechanisms, generates specialized code in *retargetable* virtual machine languages called VCODE and ICODE[26].

[3] Reconciling Efficiency and Portability

The first problem suggests that: (1) RTS systems should have specialization-time optimizations to achieve effective specialization, and (2) those optimizations should be fast enough so that their overheads can be easily amortized.

Implementing such optimizations from scratch is extremely difficult because:

- There are many optimization algorithms. It is a hard task to incorporate many of them into an RTS system.
- Most optimization algorithms are usually designed for applying at compile-time. Developing their faster implementation could be difficult.
- Some optimization algorithms are dependent on architecture of an execution platform.

Instead of developing dedicated optimizers for an RTS system, a better solution would have fast and effective optimizer independent of the RTS system, and let the RTS system use those optimizers as an external module of the system. The next section introduces our Bytecode Specialization approach, which is based on those observations.

§3 Run-time Bytecode Specialization

3.1 Overview

Our proposed *run-time bytecode specialization* (BCS) technique uses a virtual machine (bytecode) language as its source and target languages. It takes a bytecode program as its input, and constructs a specializer in the same bytecode language. At run-time, the specializer, which runs on a virtual machine, generates specialized programs in the same bytecode language. As a virtual machine language, we choose the Java virtual machine language (JVML)[21].

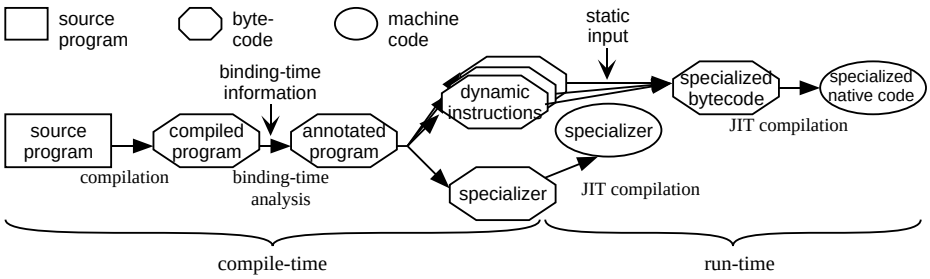

Fig. 2 Overview of BCS.

We aim to solve the problems in the previous section in the following ways:

Efficiency. Instead of directly generating specialized code in a native machine language, BCS generates it in an intermediate (bytecode) language. When the system is running on a JVM with a JIT compiler, the specialized code is optimized into a native machine language before execution.

Another functionality of specializers in BCS is that they can perform method inlining at run-time. Although the specialized code with method inlining has a certain amount of overheads for saving/restoring local variables at bytecode level, a JIT compiler can remove most of them according to our experiments.

Portability. As is shown in a previous run-time code generation system[11, 26], code generation at the virtual machine level can improve portability. Current BCS system generates specialized code in the standard JVML; the generated code can be executed on JVMs that are a widely available to various platforms.

The input to the BCS system is a JVML program. This means that the system does not depend on the syntax of high-level languages. Instead, run-time specialization can be applied to any language for which there exists a compiler into JVML. In fact, there are several compilers from various high-level languages to JVML[7, 4, etc.], which would be used as a front-end when we extended our system to support the fullset of JVML.

As shown in Fig. 2, a compiler first translates a source program written

in a high-level language (*e.g.,* Java) into JVML bytecode. The compiled program is annotated by using our binding-time analysis algorithm. From the annotated program, a specializer for generating the *dynamic instructions* is constructed. At run-time, the specializer takes the values for the *static parameters* and generates a specialized program in bytecode by writing the dynamic instructions in an array. Finally, the JVM's class loader and the JIT compiler translate the bytecode specialized program into machine code, which can be executed as a method in the Java language.

In the following subsections, we present an outline of each process in BCS briefly.

3.2 Source and Target Language

As mentioned, our source and target language is JVML, which is a stack-machine language with local variables and instructions for manipulating objects. Currently, a subset of the JVML instructions is supported. Restrictions are:

- Only primitive types are supported. (*i.e.,* reference types such as arrays and objects are not yet supported[*2].)
- All methods must be *class methods* (*i.e.,* methods are declared `static`).
- Subroutines (`jsr` and `ret`), exceptions, and multi-threading are not supported.

Fig. 3 shows the result of compiling method `power` (Fig. 1) into JVML. A method invocation creates a *frame* that holds an *operand stack* and *local variables*. An instruction first pops zero or more values off the stack, performs computation, and pushes zero or one value onto the stack.

The `iconst` n instruction pushes a constant n onto the stack. The `isub` (or `imul`) instruction pops two values off the stack and pushes the difference (or multiple) of them onto the stack. The `iload` x instruction pushes the current value of local variable x onto the stack. The `istore` x instruction pops a value off the stack and assigns it to local variable x. The `ifne` L instruction pops a value off the stack and jumps to address L in the current method if the value is not zero. The `invokestatic` $t_0\ m(t_1, \ldots, t_n)$ instruction invokes method m with the first n values on the stack as arguments. The `invokestatic` instruction (1) pops n values off the stack, (2) saves the current frame and program counter, (3) assigns the popped values into variables $0, \ldots, (n-1)$ in a newly allocated

[*2] Our current prototype system can yet treat static arrays of primitive values in a restricted context. A specialized program is correct only when arrays are modified under a static control flow with no aliases.

```
Method int Power.power(int,int)
 0 iload 1                                  // push n
 1 ifne 4                                   // go to 4 if n ≠ 0
 2 iconst 1                                 // (case n = 0)push 1
 3 ireturn                                  // return 1
 4 iload 0                                  // (case n ≠ 0)push x
 5 iload 0                                  // push x as arg. #0
 6 iload 1                                  // push n
 7 iconst 1                                 // push 1
 8 isub                                     // compute (n − 1) as arg. #1
 9 invokestatic int Power.power(int,int)    // call method
10 imul                                     // compute x × (return value)
11 ireturn                                  // return x × (return value)
```

Fig. 3 Method `power` in JVML.

frame, and (4) jumps to the first address of method m. The `ireturn` instruction (1) pops a value off the stack, (2) disposes of the current frame and restores the saved one, (3) pushes the value on the restored stack, and (4) jumps to the next address of the saved program counter. The caller uses the value at the top of the stack as a returned value.

3.3 Binding-Time Analysis

Our binding-time analysis algorithm is a flow sensitive and monovariant (context insensitive) analysis for the subset of JVML based on a type system. From the viewpoint of binding-time analysis, the subset of JVML is mostly similar to high-level imperative languages such as C. Therefore, the analysis should be careful about the following respects, unlike the analyses for functional languages:

- Since compilers may assign different variables to an operand-stack entry or a local variable, the analysis should be *flow sensitive*[17]; *i.e.,* it should allow an operand-stack entry or a local variable to have a different binding-time at each program point in a method.
- As JVML is an *unstructured* language (*i.e.,* it has a 'goto' instruction), merge points of a conditional jump and loops are implicit. The algorithm therefore has to somehow infer this information.

The binding-time analysis algorithm is based on a type system, following

the algorithms used for functional languages[2, 16]. As the type system, we use a type system of JVML proposed by Stata and Abadi[31)*3]. The algorithm, which is described in Appendix 1, consists of the following steps:

1. A preprocessor modifies the control flow of the given program so that any conditional jump has at least one merge point within the method. Specifically, it inserts a unique `ireturn` instruction at the end of the method, and replace all `ireturn` instructions with `goto` instructions to the inserted `ireturn` instruction.
2. In a given program, for each address in each method, it first gives three type variables to an operand-stack, to a frame of local variables, and to an instruction at the address. By giving different type variables to local variables at each address, the system achieves flow sensitivity, as well as the original Stata and Abadi's system.
3. It then applies typing rules to each instruction of a method, and generates constraints among the type variables.
4. It also generates additional constraints that treat *non-local side-effects under dynamic control*[19, chapter 11] by using the result of a flow analysis.
5. It finally computes a minimal set of assignments to type variables that satisfies all the generated constraints.

Example Fig. 4 shows a binding-time annotated `power` when the binding-times of `x` and `n` are dynamic and static, respectively. The binding-time of an instruction, which is displayed in the B column, is either S (static) or D (dynamic). The binding-time of a stack, which is displayed in the T column, is written as $\tau_1 \cdot \tau_2 \cdots \tau_n \cdot \epsilon$ (a stack with n values whose types are $\tau_1,\ \tau_2, \ldots$, from the top value). The binding-time of a frame of local variables, which is displayed in the F column, is denoted as $\emptyset$ (an empty frame) or $[i_k \mapsto \tau_k]$ (a frame whose local variable i_k has type τ_k). Note that the domains of the frame types 'shrink' along the execution paths. This is because our binding-time analysis rules generate constraints on only types of live local variables, and the types of unused ones do not appear in the result.

The result of the binding-time analysis is effectively the same as that of the source-level binding-time analysis; *i.e.,* instructions that correspond to a

*3 They design their type system for formalizing the JVM's verification rules in terms of subroutines (`jsr` and `ret`). Here, our current analysis merely uses the style of their formalization, and omits complicated rules for subroutines.

	instruction	B	T	F
	`iload 1`	S	ϵ	$[0 \mapsto D, 1 \mapsto S]$
	`ifne L2`	S	$S \cdot \epsilon$	$[0 \mapsto D, 1 \mapsto S]$
`L1 :`	`iconst 1`	S	ϵ	$\emptyset$
	`goto L0`	S	$D \cdot \epsilon$	$\emptyset$
`L2 :`	`iload 0`	D	ϵ	$[0 \mapsto D, 1 \mapsto S]$
	`iload 0`	D	$D \cdot \epsilon$	$[0 \mapsto D, 1 \mapsto S]$
	`iload 1`	S	$D \cdot D \cdot \epsilon$	$[1 \mapsto S]$
	`iconst 1`	S	$S \cdot D \cdot D \cdot \epsilon$	$\emptyset$
	`isub`	S	$S \cdot S \cdot D \cdot D \cdot \epsilon$	$\emptyset$
	`invokestatic`	S	$S \cdot D \cdot D \cdot \epsilon$	$\emptyset$
	`int Power.power(int,int)`			
	`imul`	D	$D \cdot D \cdot \epsilon$	$\emptyset$
	`goto L0`	S	$D \cdot \epsilon$	$\emptyset$
`L0 :`	`ireturn`	S	$D \cdot \epsilon$	$\emptyset$

Fig. 4 A binding-time annotated `power`.

static or dynamic expression at source-level have the static or dynamic types, respectively.

3.4 Specializer Construction

From an original program and a result of binding-time analysis, a specializer is constructed in "pure" JVML. It generates specialized code on a per-instruction basis at run-time[20]. For each dynamic instruction in the original program, the specializer has a sequence of instructions that writes the bytecode of the instruction into an array. The specializer also performs method inlining by successively running specializers of a method caller and callee, and by inserting a sequence of instructions that saves and restores local variables appropriately.

Here, we describe the construction of a specializer by using pseudo-instructions. Note that those pseudo-instructions are used only for explanation, and they are replaced with sequences of pure JVML instructions in the actual specializer. The specializer is executable as a Java method.

The extended JVML for defining specializers contains the JVML instructions and pseudo-instructions, namely, `GEN` *instruction*, `LIFT`, `LABEL` L, `SAVE` n $[x_0, \ldots]$, `RESTORE`, and `INVOKEGEN` m $[x_0, \ldots]$, where *instruction* is a standard JVML instruction. Fig. 5 shows an example definition of specializer `power_gen` with pseudo-instructions, constructed from method `power`. A specializer is constructed by translating each annotated instruction as follows.

- Static instruction i becomes instruction i of the specializer.

- Dynamic instruction i is translated into pseudo-instruction `GEN` i. When `GEN` i is executed at specialization time, the binary representation of i is written in the last position of an array where specialized code is stored.
- When an instruction has a different binding-time than that of the value pushed or popped by the instruction, pseudo-instruction `LIFT` is inserted. More precisely, (1) when a static instruction at program counter pc pushes a value onto the stack and $T[pc+1] = D \cdot \sigma$, where σ denotes an arbitrary stack type, `LIFT` is inserted *after* the instruction. The `iconst_1` at `L1` in Fig. 4 is an example. (2) When a dynamic instruction at pc pops a value off the stack and $T[pc] = S \cdot \sigma$, `LIFT` is inserted *before* the instruction. The execution of a `LIFT` instruction pops value n off the stack and generates instruction "`iconst` n" as an instruction of the specialized program.
- Static `invokestatic` $t_0\ m(t_1, \ldots, t_n)$ is translated into pseudo-instruction `INVOKEGEN` m`_gen`$(t_{j_1}, \ldots, t_{j_k})\ [x_0, x_1, \ldots]$, where $t_{j_1}, \ldots, t_{j_k}$ are the types of static arguments, and $x_0, x_1, \ldots$ are the dynamic local variables at the current address. When `INVOKEGEN` is executed, (1) instructions that save local variables $x_0, x_1, \ldots$ to the stack and move values on top of the stack to the local variables are generated, (2) a specializer m`_gen` is invoked, and (3) instructions that restore saved local variables $x_0, x_1, \ldots$ are generated. The number of values moved from the stack to the local variables in (1) is the number of dynamic arguments of m.
- When conditional jump `ifne` L is dynamic, the specializer has an instruction that generates `ifne`, followed by the instructions for the 'then' and 'else' branches. In other words, it generates specialized instruction sequences of both branches, one of which is selected by the dynamic condition. First, the jump instruction is translated into two pseudo-instructions: `GEN ifne` L and `SAVE` $n\ [x_0, x_1, \ldots]$, where n and $[x_0, x_1, \ldots]$ are the number of static values on the stack that will be popped during the execution of the 'then' branch and a list of static local variables that may be updated during execution of the 'then' branch, respectively. In addition, pseudo-instruction sequence `LABEL` L`; RESTORE` is inserted at label L. When `SAVE` is executed at specialization time, the top n values on the current stack and the local variables $x_0, x_1, \ldots$ are saved. The execution of `RESTORE` resets the saved values on the stack and in the frame.

```
Method Power.power_gen(int)
    iload_1
    ifne L2
 L1:iconst_1
    LIFT
    goto L0
 L2:GEN iload_0
    GEN iload_0
    iload_1
    iconst_1
    isub
    INVOKEGEN Power. power_gen(int) []
    GEN imul
    goto L0
 L0:return
```

Fig. 5 Specializer definition with pseudo-instructions.

3.5 Specializer Execution

The specializer definition is further translated into a Java method so that it takes (1) several parameters needed for specialization including an array that stores instructions of the specialized program and (2) the static arguments of the original method.

When a program uses the specializer, the following operations are performed: **(CP creation)** A '*Constant Pool*' (CP) object that records lifted values during specialization is created. **(specializer execution)** The specializer method is invoked with static arguments and the other necessary information for specialization. **(class finalization)** From the specialized instructions written in a `byte` array and the CP object, a ClassFile image[*4] is created. **(class loader creation)** A ClassLoader object is created[*5]. **(class loading)** Using the ClassLoader object, the ClassFile image is loaded into the JVM, which defines a new class with the specialized method. **(Instance creation)** An instance of

[*4] Despite its name, a ClassFile image in our system is created as a `byte` array. No files are explicitly created for class loading.

[*5] Since some JVM implementations significantly slowed down when a ClassLoader object loads a number of classes in our experiment, we create a class loader for each specialized code. Section 4.3 shows that the time for creating of a ClassLoader object is insignificant among the overall specialization overheads.

```
Method int power_2(int)
 0 iload_0
 1 iload_0
 2 istore_0
 3 iload_0
 4 iload_0
 5 istore_0
 6 iconst_1
 7 imul
 8 imul
 9 ireturn
```

Fig. 6 Specialized version of `power` with respect to 2.

the newly defined class is created. The program finally can call the specialized method via a virtual method of the instance.

Fig. 6 shows the instructions for specialized `power` with 2 as a static argument. Some instructions, such as those that load a value immediately after storing the value, are unnecessary. Those instructions arise to save/restore local variables around inlined methods.

§4 Performance Measurement

4.1 An Application Program: Mandelbrot Sets Drawer

As a target of specialization, we took a non-trivial application program that interactively displays the Mandelbrot sets. The user of the program can enter the definition of a function, and the program displays the image of the Mandelbrot set that is defined by using the function. Since the function is given interactively, the program defines an interpreter for evaluating mathematical expressions. In order to draw an image of the set, the application have to evaluate the function more than one million times. This means that run-time specialization of the interpreter with respect to a given expression could improve the performance of the drawing process.

In our performance measurements, the method `eval` and its auxiliary methods in the interpreter, which take an expression and a store, and returns the value of the expression, are specialized with respect to an expression "`z*z+c`". Since current BCS implementation does not support objects, we modified the

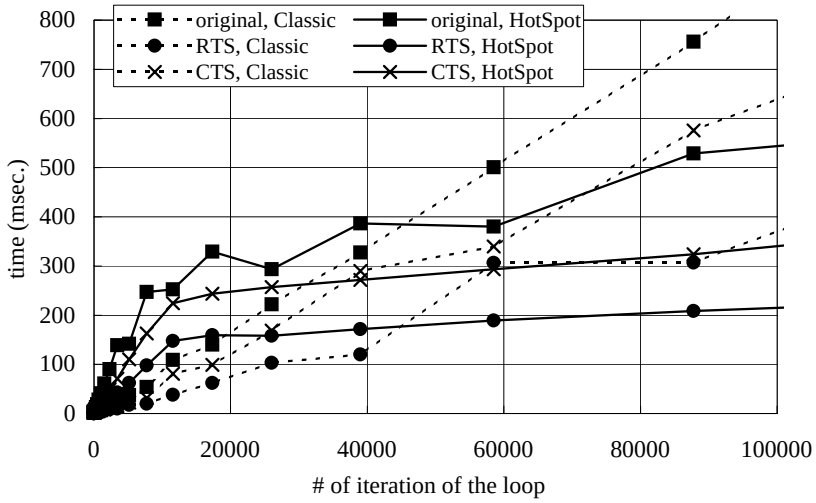

Fig. 7 Execution times of loops of specialized and unspecialized `eval`.

method to use arrays for representing expressions and stores.

We measured execution times of the target methods on two JVMs with different JIT compilers, namely, Sun "Classic" VM for JDK 1.2.1 with sunwjit compiler, and Sun "HotSpot" VM (in "client" mode) for JDK 1.2.2, in order to examine impacts of a JIT strategy on the specialization performance. All programs are executed on Sun Enterprise 4000 with 14 UltraSPARCs at 167MHz, 1.2GB memory, and SunOS 5.6. Execution times are measured by inserting `gethrvtime` system calls, which is called via a native method.

4.2 Performance of Specialized Method

We measured performance of three versions of the `eval` method on the abovementioned JVMs. The first one is the 'original' unspecialized method. The second one is a run-time specialized ('RTS') method generated by the BCS system. The third one is a compile-time specialized ('CTS') method, which is obtained by translating the original method into a C function, then applying Tempo[24], and finally translating specialized C function back into Java. At bytecode-level, the RTS and CTS methods are not identical in terms of usage of operand stack and local variables. The RTS method uses the operand stack to save/restore local variables around inlined methods. The CTS method, on the other hand, uses distinct (and thus many) local variables for the inlined methods.

Table 1 Execution times (μ seconds) of `eval` method.

	original		RTS			CTS	
VM	B_O	J_O	B_R	J_R	S	B_C	J_C
Classic	6.405	2,513	2.255	1,330	2,658	2.257	1,792
HotSpot	2.774	245,156	0.691	146,795	3,129	0.659	159,688

Fig. 7 shows the execution times of the method, which are measured by the following way. A ClassLoader object in our benchmark program first loads a new class that contains the (either specialized or unspecialized) `eval` method. The program then measures execution time of a loop that repeatedly invokes the `eval` method. Note that the measured time does *not* include specialization process, but *does* include the time of JIT compilation processes because JVMs perform JIT compilation during method invocations. As a result, the curves of the graph are not linear for small iteration numbers.

We therefore estimated, for each curve, execution times of the JIT-compiled body of the method (hereafter referred to as B) and JIT compilation process (J), by using an linear approximation of the curve at large iteration numbers.

Table 1 shows the estimated execution times of the method. It also shows specialization overheads (S), which are explained in the next subsection. As we see in the J_O, J_R and J_C columns, JIT compilation processes took from one millisecond to a few hundred milliseconds, depending on the JIT compilers. Table 2 shows the relative speed. As we see in the B_O/B_R and B_R/B_C columns, the run-time specialized code runs 3–4 times faster than the unspecialized one does, and achieves almost the same speedup factors as the compile-time specialized code does.

Table 3 shows static properties of the specialized methods. Since the original `eval` is defined with two more auxiliary methods, the sum of the values of the three methods are displayed in the table. The 'method size' row shows the bytecode instruction size of each version. Since RTS version has instructions that save/restore local variables, it is longer than the CTS version, but uses less local variables.

4.3 Specialization Overheads and Break-even Points

Elapsed times for the specialization processes (S) are measured by av-

Table 2 Relative speeds of eval method.

VM	B_O/B_R	B_O/B_C	B_R/B_C
Classic	2.841	2.838	0.999
HotSpot	4.014	4.212	1.049

Table 3 Static properties of specialized and unspecialized methods.

	original		RTS	CTS
method size (bytes)	$(132 + 67 + 210 =)$	409	173	135
maximum stack depth	$(7 + 6 + 5 =)$	18	7	5
number of local variables	$(9 + 9 + 13 =)$	31	11	24

Table 4 Breakdown of specialization overheads *with* bytecode verifier.

VM	Classic		HotSpot	
process	time(μsec.)	(ratio)	time(μsec.)	(ratio)
CP creation	46.38	(1.7%)	95.91	(3.1%)
specializer execution	61.67	(2.3%)	194.81	(6.2%)
class finalization	55.77	(2.1%)	125.18	(4.0%)
class loader creation	16.68	(0.6%)	22.14	(0.7%)
class loading	1,907.33	(71.8%)	1,518.18	(48.5%)
instance creation	569.73	(21.4%)	1,172.96	(37.5%)
total (S)	2,657.57	(100.0%)	3,129.19	(100.0%)

Table 5 Breakdown of specialization overheads *without* bytecode verifier.

VM	Classic		HotSpot	
process	time(μsec.)	(ratio)	time(μsec.)	(ratio)
CP creation	46.24	(3.3%)	89.44	(3.8%)
specializer execution	66.27	(4.7%)	218.80	(9.2%)
class finalization	37.16	(2.7%)	103.65	(4.3%)
class loader creation	12.59	(0.9%)	19.18	(0.8%)
class loading	776.16	(55.7%)	1,708.72	(71.7%)
instance creation	455.00	(32.7%)	243.69	(10.2%)
total	1,393.43	(100.0%)	2,383.49	(100.0%)

Table 6 Break-even points.

VM	verifier	over JIT compiled code	over newly loaded code
Classic	on	961	355
	off	656	50
HotSpot	on	71,975	(less than zero)
	off	71,617	(less than zero)

eraging 10,000 runs. Table 4 and Table 5 shows the times for each sub-process, which is explained in Section 3.5. Table 4 is the result of execution on JVMs *with* bytecode verifiers, and Table 5 is on JVMs *without* verifiers (by using undocumented "`-Xverify:none`" option of Sun's JDK). As we see, 80–90% time of the specialization process is spent for the ones inside the JVM, namely, class loading and instance creation. More specifically, 25–50% of the specialization process is spent for the JVM's bytecode verification as the differences of numbers between Table 4 and Table 5 suggest.

We presume that some of overheads could be removed if we integrated our system with a JIT compiler so that the specializer directly generates specialized code in an intermediate representation in the JIT compiler.

A *break-even point* (BEP) is a number of runs of a specialized program needed to amortize the specialization cost over the execution time of the unspecialized program. In programming systems that perform dynamic optimizations, even unspecialized programs have to pay overheads of the optimization, namely JIT compilation time. We therefore calculated two BEPs. The first one assumes that the unspecialized code is already JIT compiled. In this case, a BEP, which is calculated by the formula $(J_R + S)/(B_O - B_R)$, is approximately 700–72,000 runs as shown in Table 6. The second one assumes that the unspecialized code is newly loaded, and thus pays the cost of JIT compilation during its execution. The BCS specialized code exhibits a small BEP in this case, which is computed by the formula $(J_R + S - J_O)/(B_O - B_R)$. Note that the benchmark application, in order to draw an image of a given expression, executes the `eval` method for much larger number of times than the BEPs. This means that BCS actually improves the overall execution times of the application.

4.4 Comparison to a Native Code Run-time Specialization System

Table 7 Execution and specialization times and break-even points of `eval` in Tempo.

execution times (μsec.)				relative speed			BEP
original	RTS		CTS				
B_O	B_R	S	B_C	B_O/B_R	B_O/B_C	B_R/B_C	
1.318	0.675	23.829	0.311	1.95	4.24	2.17	37.1

In order to compare the speedup factors and specialization overheads with a run-time native-code specialization system, we also wrote the same interpreter in C, and specialized by using Tempo 1.194[*6]. The interpreter is compiled by GCC 2.7.2 with `-O2` option. All the other execution environments are the same to the previous ones.

Table 7 shows the execution times and specialization times that are measured by averaging ten million runs. We observe that the run-time specialized code is slower than the compile-time specialized one in Tempo[*7].

Comparing between the execution time in BCS and the one in Tempo, we notice that compile-time specialized codes in those two systems show the similar speedup factors (B_O/B_C). On the other hand, the speedup factors of the run-time specialized code (B_O/B_R) in Tempo are worse than the one in BCS. This can be an evidence of our premise: performing optimizations after specialization could be useful to improve performance of run-time specialized code.

§5 Related Work

Tempo is a compile-time and run-time specialization system for C language[24]. Tempo achieves portability by using outputs of standard C compilers to construct specializers. As the specializers simply copy machine instructions to memory at run-time, their BEP numbers are low (3 to 87 runs in their realistic examples). On the other hand, the specializers perform no optimizations and no function inlining at run-time specialization.

DyC is another RTS system for C language[14]. The analysis and spe-

[*6] We set both `reentrant` and `post_inlining` options of Tempo to `true`, and the compiler options for both templates and specializers to `"-O2"`. We also implemented an efficient memory allocator for residual code.

[*7] In the previous version of the paper the run-time specialized code in Tempo exhibited tremendously bad performance—more than two hundred times slower than the original code. Most of the overheads, however, turned out to be caused by a bug in the benchmark program.

cializers can directly handle unstructured C programs. The system generates highly optimized code, by developing its own optimizing compiler for Digital Alpha 21164. It can perform optimizations at run-time specialization[3]. However, the optimizations seem to make the BEP numbers larger (around 700 to 30,000), similar to BCS.

Fabius is an RTS for pure-functional subset of ML, targeting MIPS R3000[20]. Because the source language is a pure functional language, the binding-time analysis and specializer construction in Fabius are simpler than those for imperative and unstructured languages. Similar to BCS, specializers in Fabius are on a per-instruction basis and perform function inlining for tail recursive functions. It is also suggested that the specializers would perform register allocation at run-time.

Fujinami proposes a run-time specialization system for C++, targeting MIPS R4000 and Intel x86[12]. A specialized program is his system runs twice as fast as the one compiled by a statically optimizing compiler. His system achieves this speedup by embedding a number of optimization algorithms into a statically generated specializer. Our approach, on the other hand, is to optimize a specialized code by using a JIT compiler, which is an independent module.

`C is a language with dynamic code generation mechanisms[26]. Unlike other RTS systems, `C programmers have to explicitly specify binding-times of expressions. Similar to BCS, the implementation of `C generates programs in virtual machine languages called VCODE and ICODE. The run-time system of ICODE performs optimizations including register allocation for generated programs, similar to JIT compilers for JVMs.

Bertelsen proposes, independently of BCS, an algorithm for binding-time analysis of a JVML subset[6]. His algorithm can treat arrays, and does not treat objects. Also, it only targets a single method. A specialization process based on the analysis is only informally discussed.

JSpec is an off-line, compile-time partial evaluator for Java[30, 29, 28]. The system analyzes and specializes Java programs by applying Tempo, a partial evaluator for C, after translating the Java programs into C. A specialized program can be executed either by compiling the specialized C program into native code, or by translating it back into Java. This approach can be compared to ours that uses a compiler from a high-level language to a bytecode language as a front-end. Unlike current BCS implementation, JSpec supports all the object-oriented features of Java whose specialization strategies are specified through

specialization classes[32].

§6 Conclusion

In this paper, we have proposed run-time bytecode specialization (BCS), which specializes Java virtual machine language (JVML) programs at run-time. The characteristics of this approach are summarized as follows: (1) the system directly analyzes a program and creates a specializer in an intermediate language JVML; and (2) the specializer generates programs in JVML, which makes it possible to apply optimizations after specialization by using existing JVMs with just-in-time (JIT) compilers.

The binding-time analysis algorithm is based on a type system, and also uses results of flow analysis to correctly handle stacks, local variables, and side-effects.

Thus far, we have implemented a prototype BCS system for a JVML subset and have shown that a method in a non-trivial program specialized by our system runs approximately 3–4 times faster than the unspecialized method. The specialization cost can be amortized by 1,000 to 72,000 runs, depending on the JVMs. Those numbers are worse than the ones in the systems that are rather focusing on the specialization speed[20, 24], though.

We are now extending our system to support the full JVML. Since current implementation only supports primitive types, rules that properly handle references to objects and arrays should be devised. To support objects and arrays, the system needs information whether data is modified by other methods or other threads. Such information could be obtained by either static analysis (*e.g.,* the one studied by Choi, et al.[8]) or through user declarations which could be written in the style of specialization classes[32]. In practice, it is also important to support other features, such as multi-threading, and subroutines (*i.e.,* `jsr` and `ret` instructions in JVML) and exceptions. Some may consider that templates of bytecode would reduce specialization costs. As our experiments in Section 4 showed, however, the major sources of specialization overheads are class loading and JIT-compilation. Rather than improving the performance of the bytecode generation process, our current plan is to generate a specialized program directly in an intermediate language of a JIT compiler, by using JVMs with interfaces to JIT compilers[25].

Acknowledgment We are grateful to Julia Lawall for her advice on

benchmark executions in Tempo. We would like to thank Reynald Affeldt, Naoya Maruyama, and Satoshi Matsuoka for discussing on the BCS system. We also would like to thank Robert Glück, Olivier Danvy, Ulrik Schultz, Lars Clausen for their comments. Finally, we are very grateful to the detailed and constructive comments from anonymous reviewers on the early draft of the paper.

References

1) ACM. *Proceedings of Conference on Programming Language Design and Implementation (PLDI'96)*, volume 31(5) of *ACM SIGPLAN Notices*, Philadelphia, PA, May 1996.

2) Asai, K. "Binding-Time Analysis for Both Static and Dynamic Expressions". In *SAS'99*, volume 1694 of *Lecture Notes in Computer Science*, 1999.

3) Auslander, J., Philipose, M., Chambers, C., Eggers, S. J., and Bershad, B. N. "Fast, Effective Dynamic Compilation". In *Proceedings of Conference on Programming Language Design and Implementation (PLDI'96)* [1)], pp. 149–159.

4) Benton, N., Kennedy, A., and Russell, G. "Compiling standard ML to Java bytecodes". In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, Baltimore, MD, September 1998.

5) Berlin, A. A. and Surati, R. J. "Partial Evaluation for Scientific Computing: The Supercomputer Toolkit Experience". In *Partial Evaluation and Semantics-Based Program Manipulation*, pp. 133–141, Orlando, FL, June 1994. ACM SIGPLAN. Published as Technical Report 94/9, Department of Computer Science, University of Melbourne.

6) Bertelsen, P. "Binding-time analysis for a JVM core language". ftp://ftp.dina.kvl.dk/pub/Staff/Peter.Bertelsen/bta-core-jvm.ps.gz, April 1999. http://www.dina.kvl.dk/ pmb/.

7) Bothner, P. "Kawa – Compilinig Dynamic Languages to the Java VM". In *USENIX*, New Orleans, June 1998.

8) Choi, J.-D., Gupta, M., Serrano, M., Sreedhar, V. C., and Midkiff, S. "Escape Analysis for Java". In Northrop, L. M., editor, *Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*, volume 34(10) of *ACM SIGPLAN Notices*, pp. 1–19, Denver, Colorado, USA, October 1999. ACM.

9) Consel, C. and Noël, F. "A General Approach for Run-Time Specialization and its Application to C". In *Conference Record of Symposium on Principles of Programming Languages (POPL96)*, pp. 145–170, St. Petersburg Beach, Florida, January 1996. ACM SIGPLAN-SIGACT.

10) Danvy, O. and Filinski, A., editors. *Second Symposium on Programs as Data Objects (PADO II)*, volume 2053 of *Lecture Notes in Computer Science*, Aarhus, Denmark, May 2001. Springer-Verlag.

11) Engler, D. R. "VCODE: A Retargetable, Extensible, Very Fast Dynamic Code Generation System". In *Proceedings of Conference on Programming Language Design and Implementation (PLDI'96)* [1)], pp. 160–170.

12) Fujinami, N. "Determination of Dynamic Method Dispatches Using Run-Time Code Generation". In Leroy, X. and Ohori, A., editors, *Types in Compilation (TIC'98)*, volume 1473 of *Lecture Notes in Computer Science*, pp. 253–271, Kyoto, Japan, March 1998. Springer-Verlag.

13) Futamura, Y. "Partial evaluation of computation process—an approach to a compiler-compiler". *Higher-Order and Symbolic Computation*, 12, 4, pp. 381–391, 1999. Reprinted from *Systems, Computers, Controls*, 2, 5, pp. 45–50, 1971.

14) Grant, B., Philipose, M., Mock, M., Chambers, C., and Eggers, S. J. "An Evaluation of Staged Run-Time Optimization in DyC". In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*, 1999.

15) Guenter, B., Knoblock, T. B., and Ruf, E. "Specializing Shaders". In *SIGGRAPH'95 (Computer Graphics Proceedings, Annual Conference Series)*, pp. 343–349, 1995.

16) Henglein, F. "Efficient Type Inference for Higher-Order Binding-Time Analysis". In *FPCA'91 Proceedings of Functional Programming Languages and Computer Architecture (LNCS 523)*, volume 523 of *Lecture Notes in Computer Science*, pp. 448–472, 1991.

17) Hornof, L. and Noyé, J. "Accurate Binding-Time Analysis for Imperative Languages: Flow, Context, and Return Sensitivity". In *Proceedings of Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97)*, volume 32(12) of *ACM SIGPLAN*, pp. 63–73, Amsterdam, June 1997. ACM.

18) Hornof, L. and Jim, T. "Certifying Compilation and Run-time Code Generation". In Danvy, O., editor, *Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, volume NS-99-1 of *BRICS Notes Series*, pp. 60–74, San Antonio, Texas, January 1999. ACM SIGPLAN.

19) Jones, N. D., Gomard, C. K., and Sestoft, P. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

20) Lee, P. and Leone, M. "Optimizing ML with Run-Time Code Generation". In *Proceedings of Conference on Programming Language Design and Implementation (PLDI'96)* [1], pp. 137–148.

21) Lindholm, T. and Yellin, F. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

22) Masuhara, H. and Yonezawa, A. "Design and Partial Evaluation of Meta-objects for a Concurrent Reflective Language". In Jul, E., editor, *European Conference on Object-Oriented Programming (ECOOP'98)*, volume 1445 of *Lecture Notes in Computer Science*, pp. 418–439, Brussels, Belgium, July 1998. Springer-Verlag.

23) Masuhara, H. and Yonezawa, A. "Run-time Bytecode Specialization: A Portable Approach to Generating Optimized Specialized Code". In Danvy and Filinski [10], pp. 138–154.

24) Noël, F., Hornof, L., Consel, C., and Lawall, J. L. "Automatic, Template-Based Run-Time Specialization: Implementation and Experimental Study". In *IEEE International Conference on Computer Languages (ICCL '98)*, pp. 123–142, Chicago, Illinois, USA, May 1998.

25) Ogawa, H., Shimura, K., Matsuoka, S., Maruyama, F., Sohda, Y., and Kimura, Y. "OpenJIT: An Open-Ended, Reflective JIT Compile Framework for Java". In Bertino, E., editor, *14th European Conference on Object-Oriented Programming (ECOOP 2000)*, volume 1850 of *Lecture Notes in Computer Science*, pp. 362–387, Sophia Antipolis and Cannes, France, June 2000. Springer-Verlag.

26) Poletto, M., Hsieh, W. C., Engler, D. R., and Kaashoek, M. F. "'C and tcc: a Language and Compiler for Dynamic Code Generation". *Transactions on Programming Languages and Systems*, 21, 2, pp. 324–369, March 1999.

27) Pu, C., Autrey, T., Black, A., Consel, C., Cowan, C., Inouye, J., Kethana, L., Walpole, J., and Zhang, K. "Optimistic incremental specialization: streamlining a commercial operating system". In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, pp. 314–324, Copper Mountain Resort, Colorado, USA, December 1995.

28) Schultz, U. *Object-Oriented Software Engineering using Partial Evaluation.* PhD thesis, University of Rennes, December 2000. English version available as IRISA PI 1395.

29) Schultz, U. "Partial Evaluation for Class-Based Object-Oriented Languages". In Danvy and Filinski [10], pp. 173–197.

30) Schultz, U. P., Lawall, J. L., Consel, C., and Muller, G. "Towards Automatic Specialization of Java Programs". In Guerraoui, R., editor, *European Conference on Object-Oriented Programming (ECOOP'99)*, volume 1628 of *Lecture Notes in Computer Science*, pp. 367–390, Lisbon, Portugal, June 199. Springer-Verlag.

31) Stata, R. and Abadi, M. "A Type System for Java Bytecode Subroutines". *Transactions on Programming Languages and Systems*, 21, 1, pp. 90–137, January 1999.

32) Volanschi, E., Consel, C., Muller, G., and Cowan, C. "Declarative Specialization of Object-Oriented Programs". In Bloom, T., editor, *Proceedings of Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 32 (10) of *ACM SIGPLAN Notices*, pp. 286–300, Atlanta, October 1997.

33) Wickline, P., Lee, P., and Pfenning, F. "Run-time Code Generation and Modal-ML". In *Proceedings of Conference on Programming Language Design and Implementation (PLDI'98)*, volume 33(5) of *ACM SIGPLAN Notices*, pp. 224–235, Montreal, Canada, June 1998. ACM.

§1 Rules for Binding-Time Analysis

The typing rules use S and D, respectively, for static and dynamic values. The instruction types are also S and D; the sub-typing relation between them is $S \leq D$. A stack type is a string of value types, either ϵ or $\alpha \cdot \sigma$, where ϵ is the empty stack type, α is a value type, and σ is a stack type. A frame type is the map from a local variable to a value type. An extension of a value f of

frame type is defined as

$$(f[x \mapsto \alpha])[y] \equiv \text{if } x = y \text{ then } \alpha, \text{ else } f[y],$$

where x and y are local variables and α is a value type. An empty map is written as ϕ, and we abbreviate $\phi[x_1 \mapsto \alpha_1][x_2 \mapsto \alpha_2] \cdots$ as $[x_1 \mapsto \alpha_1, x_2 \mapsto \alpha_2, \ldots]$. Program P is a vector of instructions indexed by program counters. Program counter pc is a pair of method name m and instruction address i in a method; it can be written as $\langle m, i \rangle$. An increment of program counter is defined as

$$\langle m, i \rangle + 1 \equiv \langle m, i + 1 \rangle \quad .$$

Initial binding-time assignment I is given as a map from a method name to a frame type.

Program P has valid binding-times with respect to initial binding-time assignment I if there are vectors A, R, B, F, and T that satisfy the following judgment:

$$A, R, B, F, T \vdash P, I \quad ,$$

where A, R, B, F, and T are the types of method arguments, return values, instructions, frames, and stacks, respectively.

The rule to prove the judgment is defined as the judgment for all methods, and constraints of I:

$$\frac{\forall m \in \mathit{Methods}(P).\ A, R, B, F, T, m \vdash P, \quad \forall m' \in \mathit{Dom}(I).I[m'] \subseteq F[\langle m', 0 \rangle]}{A, R, B, F, T \vdash P, I} \quad ,$$

where $\mathit{Dom}(I)$ is the domain of map I, $\mathit{Methods}(P)$ is a set of method names in P, and the inclusion between maps is defined as

$$\frac{\forall x \in \mathit{Dom}(f).\ f[x] = f'[x]}{f \subseteq f'} \quad .$$

The judgment of a method is defined by the rule

$$\frac{\forall i \in \mathit{Addresses}(P, m).\ A, R, B, F, T, \langle m, i \rangle \vdash P \quad T_{\langle m,0 \rangle} = \epsilon \quad F_{\langle m,0 \rangle} = A_m}{A, R, B, F, T, m \vdash P} \quad ,$$

where $\mathit{Addresses}(P, m)$ is a set of all instruction addresses of method m in program P. The first hypothesis is a local judgment applied to each instruction address in a method. The second and third are the initial conditions on the stack and the frame.

$$\frac{P[pc] = \texttt{iconst}\ n \quad F_{pc+1} \subseteq F_{pc} \quad T_{pc+1} = \alpha \cdot T_{pc} \quad B[pc] \leq \alpha}{A, R, B, F, T, pc \vdash P}$$

$$\frac{P[pc] = \texttt{iadd} \quad F_{pc+1} \subseteq F_{pc} \quad T_{pc} = \alpha \cdot B[pc] \cdot \sigma \quad T_{pc+1} = \beta \cdot \sigma \quad \alpha \leq B[pc] \leq \beta}{A, R, B, F, T, pc \vdash P}$$

$$\frac{P[pc] = \texttt{iload}\ x \quad F_{pc+1} \subseteq F_{pc} \quad T_{pc+1} = \alpha \cdot T_{pc} \quad F_{pc}[x] = B[pc] \leq \alpha}{A, R, B, F, T, pc \vdash P}$$

$$\frac{P[pc] = \texttt{istore}\ x \quad F_{pc+1} \subseteq F_{pc}[x \mapsto B[pc]] \quad \alpha \cdot T_{pc+1} = T_{pc} \quad \alpha \leq B[pc]}{A, R, B, F, T, pc \vdash P}$$

$$\frac{P[pc] = \texttt{ifne}\ L \quad F_{pc+1} \subseteq F_{pc}, \quad F_L \subseteq F_{pc} \quad \alpha \cdot T_L = \alpha \cdot T_{pc+1} = T_{pc} \quad \alpha \leq B[pc]}{A, R, B, F, T, pc \vdash P}$$

$$\frac{P[pc] = \texttt{invokestatic}\ m \quad F_{pc+1} \subseteq F_{pc} \quad n = \text{arity}(m) \quad T_{pc} = A_m[n-1] \cdots A_m[0] \cdot \sigma \quad T_{pc+1} = \alpha \cdot \sigma \quad R[m] \leq \alpha}{A, R, B, F, T, pc \vdash P}$$

$$\frac{P[pc] = \texttt{ireturn} \quad T_{pc} = \alpha \cdot \sigma \quad \alpha \leq R[m] \quad pc = \langle m, i \rangle}{A, R, B, F, T, pc \vdash P}$$

Fig. 8 Typing rules for binding-time analysis

Fig. 8 shows rules[*8] for a judgment $A, R, B, F, T, pc \vdash P$, which are explained as follows.

- When the instruction at pc pushes a value onto the stack, the type of the top value of the stack at the next address is larger than or equal to the instruction's type. Those instructions include `iconst`, `iadd`, `iload`, and `invokestatic`.
- When the instruction at pc pops values off the stack, the type of the instruction is larger than or equal to the types of values on the stack at pc. Those instructions include `iadd`, `istore`, and `ifne`.
- When the control moves from pc to pc', types of live local variables are propagated—*i.e.,* a type of a local variable at pc' must be the same as that of the variable after executing the instruction at pc. This is represented by a constraint among frame types, like $F_{pc+1} \subseteq F_{pc}$. For a jump instruction (`ifne`), the frame types at the current and jump addresses are also constrained by the same inclusion.
  When the instruction reads or writes a local variable, the type of the instruction and that of the variable are constrained[*9].
- When the instruction at pc is `invokestatic`, the argument types of the target method and the types of values on the current stack are the same, and the return type of the method and type to the top value at the next address of the `invokestatic` instruction are the same. Since our rules require that a method have only one combination of argument types, the binding-time analysis is *monovariant*.
- When the instruction at pc is `ireturn`, the value on top of the current stack has the same type as the return type of the current method.

Side-effects: The typing environments that satisfy the above rules are consistent in terms of original execution order. That is, when an annotated program is executed disregarding the annotations, the program does not "go wrong." However, a specializer, which is constructed from an annotated program, does not have exactly the same execution order as the original program—it executes both branches of a dynamic conditional jump. As a result, the specializer may go wrong with side-effecting operations.

[*8] The rules are defined for seven typical JVM instructions. The rules for the other instructions such as "`dup`" can be easily derived by combining those displayed rules.

[*9] The current rules prohibit 'lifting' the values of local variables for simplicity.

For example, consider a specializer constructed from the following method, specifying that `x` and `y` are dynamic and static, respectively:

```
int f(int x, int y) {
    if (x > 0)
        y = y + 1;
    else
        y = y * 2;
    return g(x, y / 2);
}
```

Since `y` is static, our typing rules give static types to the expressions `y=y+1`, `y=y*2`, and `y/2`. However, a correct binding-time analysis would give a dynamic type `y/2` because the value of `y` after the `if` statement depends on the value of `x`, which is dynamic.

We solve this problem by adding the following constraints. Assume $P[pc] = \texttt{if}\ L$. Let M_{pc} be the set of addresses at which execution paths from $pc+1$ and L merge. For each $pc' \in M_{pc}$, let $l_{pc'}$ be the set of local variables that are updated by the `istore` instruction during the execution paths from $pc+1$ to pc' and L to pc', and $n_{pc'}$ be the maximum number of stack entries at pc' that are pushed during the execution paths from $pc+1$ to pc' and L to pc'. The following constraint requires that the local variables and stack entries that are computed in a branch of a dynamic conditional jump be dynamic after the merger of its branches:

$$\forall pc' \in M_{pc}.\quad T_{pc'} = \beta_1 \cdots \beta_{n_{pc'}} \cdot \sigma,\quad B[pc] \leq \beta_1\ (i = 1, \ldots, n_{pc'}) \quad \forall x \in l_{pc'}.B[pc] \leq F_{pc'}[x],$$

where the second constraint specifies that the top $n_{pc'}$ stack entries at pc' have larger types than that of the instruction at pc, and the third specifies that the updated local variables have larger types than that of the instruction at pc.