

Method Inlining, Dynamic Class Loading, and Type Soundness^{*}

Neal Glew¹ and Jens Palsberg²

¹ Intel Corporation, Santa Clara, CA 95054, USA, aglew@acm.org

² UCLA Computer Science Dept, Los Angeles, CA 90095, palsberg@ucla.edu

Abstract. Method inlining is an optimisation that can be invalidated by later class loading. A program analysis based on the current loaded classes might determine that a method call has a unique target, but later class loading could add targets. If a compiler speculatively inlines methods based on current information, then it will have to undo the inlining when later classes invalidate the assumptions. The problem with undoing inlining is that it might be executing at the time of undo and therefore require a complicated, on-the-fly update of the program state. Previous work presented techniques for dealing with invalidation including on-stack replacement, preexistence analysis, extant analysis, and code patching. Until now, it has been an open question whether such operations can be done in a type-safe manner, and no formal proof of correctness exists in the literature.

In this paper we present a framework for reasoning about method inlining, dynamic class loading, and type soundness. Our example language has both a nonoptimising and an optimising semantics. At the point of dynamically loading a class, the optimising semantics does a whole-program analysis, inlines calls in the newly loaded code, and patches the invalidated inlinings in all previously loaded code. The patching is based on a new construct that models in-place update of machine code—a technique used by some virtual machines to do speculative method inlining. The two semantics are equivalent and both have a type soundness property—proving correctness and type safety of the optimisation. Our framework can form the basis of virtual machines that represent optimised code in a typed low-level language.

1 Introduction

Devirtualisation and consequent method inlining is an important, perhaps crucial, optimisation for object-oriented languages. Previous work has explored inlining for whole programs (see, e.g., [4]) and for programs that use dynamic class loading (see, e.g., [9, 3, 1, 5, 12]). That work has devised analysis and transformations for devirtualisation and method inlining, shown the effectiveness of these optimisations, and argued informally for their correctness.

When a method inlining is invalidated by class loading, it is necessary to *revirtualise* the call, that is, intuitively, replace the inlined code with the call itself. Intuitively, devirtualisation and revirtualisation are inverses of each other. Our goal is to show that while existing revirtualisation techniques are usually not phrased for typed intermediate languages, revirtualisation can be type safe. We will prove the correctness of a well-known revirtualisation transformation in the presence of dynamic class loading, and we will prove that the transformation preserves typeability. As a further contribution, we work with a typed framework for dynamic class loading that may be of independent interest.

At the end of this section we will describe our result in more detail. First we review the issues with dynamic class loading, and describe the problems of proving correctness and preserving typeability.

1.1 Dynamic Class Loading

To review the issues with devirtualisation in the presence of dynamic class loading, consider this example program in Java:

^{*} Extended abstract

```

class B {
    void m() { ... }
}
class C extends B {
    void m() { ... }
}

String cName;
B x;
if (...) { x = new C(); }
else {
    x = (B)(Class.forName(cName).newInstance());
}
x.m();

```

The expression `Class.forName(cName).newInstance()` loads a class with name `cName` and then instantiates it. The program uses class loading and it type checks. Suppose now that we do a flow analysis of the program and determine that the call site `x.m()` currently has a unique receiver, namely the `m` method in class `C` (we will write `C::m` for this method). We could then go ahead and inline the call to that method, knowing that later class loading might invalidate that inlining—we call this *speculative inlining*. To see how such invalidation could happen, suppose that the `else` branch gets executed and that it loads a class `D` which is a subclass of `B` and which overrides the method `m`. Now there are two possible targets for the call `x.m()` so the inlining of `x.m()` has been invalidated. At the Java level there is no easy way to patch the inlined call such that it reverts to doing a dynamic dispatch.

The main problem is that an invalidated method inlining may be in a currently executing method and therefore require a complicated, on-the-fly update of the program state. Previous work has presented techniques for minimizing the needed updates. For example, the preexistence analysis of Detlefs and Agesen [5] determines a set of methods for which on-the-fly changes to the stack will not be needed. Additionally, the extant analysis of Sreedhar, Burke, and Choi [12] determines a set of method inlinings which cannot be invalidated by class loading. Still, aggressive inlining and class loading will inevitably require updates of the code memory, the stack memory, or both. Our goal is to show how a particular form of such operations can be done in a type-safe manner, and to prove its correctness.

Some virtual machines (for example [9] and ORP [2,3]) use a *patching* technique, which is a form of in-place code modification to revert to unoptimised code. The basic idea is to compile the call `x.m()` to the following code:

```

...
label 11:
    [Inline x.C::m()]
label 13:
...
[out of line]
label 12:
    x.m();
    jump 13;

```

(Where out of line means after the end of the function being compiled.) Then if a class is loaded that invalidates the inlining, the virtual machine writes a `jump 12;` instruction at address 11. The virtual machine keeps datastructures on the side to store the list of assumptions being made, and the patches required to revirtualise when the assumptions no longer hold. So long as this write is atomic and properly ordered with respect to the execution of other threads, this update will revirtualise the code including any existing stack frames that are executing the code.

1.2 Correctness

That the technique of speculative inlining with patching is correct is far less obvious than the correctness of inlining for whole programs. One goal of this paper is to provide a framework that allows proving the correctness of current and future inlining techniques in the presence of dynamic class loading. In particular we will prove correct speculative devirtualisation with patching to revirtualise when dynamic class loading invalidates previous assumptions.

Devising a framework for proving correctness of optimisations in the presence of dynamic class loading is not straightforward. Traditionally, such optimisations are seen as transformations of programs before they are run. The language theoretician formalises the transformation as a mapping from programs to programs and proves that a program and its transformation have the same semantics. For transformations like speculative devirtualisation with patching, transformations are done by the just-in-time compiler (JIT) when it compiles both the initial program and any dynamically loaded classes. Furthermore, dynamic class loading might trigger transformations of existing code such as patching needed to revirtualise inlinings invalidated by the newly loaded class. We cannot formalise this as mapping of programs to programs. Instead we view optimisation as a second semantics for programs. This optimising semantics includes the transformations of classes and patching of code as part of its definition. Correctness is then proving that this semantics and the standard semantics give the same meaning to a program.

1.3 Type Preservation

Security and reliability of software is becoming increasingly more important in the internet age—just witness the spate of recent worms ravaging the internet. One way to increase the reliability (and contribute to the security) of the compilation process is to typecheck its output. For languages like Java and CLI that are implemented with a virtual machine that uses JITs, such typechecking could be done on the output of the JITs, thus partially or fully eliminating JITs from the trusted computing base. As JITs are becoming increasingly sophisticated, employing advanced optimisations such as method inlining and escape analysis, checking their output provides significant benefits to reliability and security.

All of this presupposes that JITs can produce output that typechecks, and in particular that devirtualisation can be done in a typeability preserving manner. Our own previous work [7] shows how traditional whole program devirtualisation techniques can be done in a typeability preserving manner without sacrificing performance. This paper and other work in this area (see [7] for references) has addressed only whole program analysis and optimisation—it does not address dynamic class loading.

Typeability preservation for speculative devirtualisation and dynamic class loading is difficult for the following reasons: As we observed in our previous paper, for the output of devirtualisation to typecheck some static type information must change to reflect the flow analysis that drives the optimisation. For example, in the above program x must change to type C for the devirtualisation of $x.m()$ to $x.C::m()$ to typecheck. If the devirtualisation is done speculatively, then this changing of static type information must be done speculatively. Newly loaded classes that invalidate the speculative assumptions must cause the static type information to change back. For example, if the `forName` is executed and produces a class that does not have $C::m$ as its m method, then x 's type must change back to B along with the revirtualisation of $x.m()$.

Formalising how all this type changing is done is challenging. Fortunately, there is one analysis that does not require type changing, as it is never more precise than the statically declared types. That analysis is Class Hierarchy Analysis (CHA [4]). Furthermore this analysis is a popular choice in JITs that do devirtualisation as it is linear time and more sophisticated analyses are cubic time, making them too expensive for JITs. This paper concentrates on CHA, and shows that speculative devirtualisation based on CHA and patching to revirtualise preserves typeability. Future work should investigate if the required type changing for speculative devirtualisation and for revirtualisation can be formalised and proven correct for the more precise analyses—we believe this is an interesting and challenging problem.

As with correctness, proving typeability preservation is not just showing that the transformation takes typeable programs to typeable programs. Instead, we show that the optimising semantics is type safe. In particular, proving the standard type-preservation lemma for the optimising semantics will require showing that speculative devirtualisation and patching transformations preserve typeability.

1.4 Our Result

We present a framework for reasoning about method inlining, dynamic class loading, and type soundness. As described above, this framework consists of two semantics for a language with dynamic class loading. One semantics is standard and follows previous work on formalising Java-like languages. The other semantics includes as part of its definition of the reduction of programs, the application of speculative devirtualisation to newly loaded classes and the application of patching to revirtualise invalidated devirtualisations. We prove correctness by showing the equivalence of these two semantics, and we prove typeability preservation by proving the type safety of the optimising semantics. Our contributions are two: first, the framework itself is the only attempt we know of to formalise and prove results about optimisation in the presence of dynamic class loading, and second, we show that speculative devirtualisation using CHA and patching is typeability preserving.

Our framework can form the basis of virtual machines that represent optimised code in a typed, low-level language. We view our work as the beginning of a foundation for type-safe just-in-time compilation, extending previous work which supports type-safe off-line compilation [10, 13, 11, 14, 6].

Our approach does method inlining in two steps: (1) change some dynamic method invocations to static method invocations and (2) inline the static method invocations. The idea is, as in previous work, that in a dynamic method dispatch $e.m(e_1, \dots, e_n)$, if all the objects that e could evaluate to are instances of classes which inherit m from a fixed class D , then the dynamic dispatch can be transformed to a static dispatch $e.D.m(e_1, \dots, e_n)$. (The expression $e.D.m(e_1, \dots, e_n)$ invokes D 's version of m on e with e_1 through e_n as arguments.) A static dispatch $e.D.m(e_1, \dots, e_n)$ can be inlined to $e'\{\mathbf{this}, x_1, \dots, x_n := e, e_1, \dots, e_n\}$ where D has for method m , body e' and parameters x_1 through x_n . This is nothing other than applying a nonstandard reduction rule at compile time, and it is straightforward to show that the rule is typability preserving. Our optimising semantics explicitly performs the first of the two steps; the other step is left implicit.

We have made a number of design choices for our framework, all compatible with current virtual-machine technology. First of all, as already described, the flow analysis used during class loading and JIT compilation is CHA. As demonstrated by Glew and Palsberg [7], CHA is the simplest flow analysis which supports type-safe method inlining in a setting without class loading. Moreover, CHA is fast and therefore an attractive choice for a JIT compiler.

Second, our framework assumes that all inlinings can be invalidated. This design choice simplifies our notation considerably. Note here that Sreedhar, Burke, and Choi [12] showed that in practice there are many inlinings that cannot be invalidated by class loading. Such stable inlinings are called *unconditionally-monomorphic* call sites. There is no conceptual problem with adding support for unconditionally-monomorphic call sites to our framework. Such an addition would lead to space savings in the generated code, but not execution-time savings.

Third, our framework uses a simple construct called **dynnew** which abbreviates the Java expression `Class.forName(...).newInstance()`, that is, an operation that loads some class and immediately instantiates it. Using this construct means that we do not need to model the result of `Class.forName(...)` and deal with objects that reify classes, simplifying the operational semantics. It is also easy to type check: it has type `Object`.

Fourth, we abstract some details of patching. In particular, we assume that it is an atomic operation. An actual implementation of the technique in a virtual machine must ensure this atomicity. It must atomically write the jump instruction, it must ensure consistency between icache and dcache, and must correctly deal with weak memory ordering on architectures with this model. Such concerns are very low-level, we abstract them into an atomic operation that is appropriate for the intermediate language level that we are considering.

Once the framework is defined, the main technical challenge is to reason about the optimising semantics. In contrast to most work on type preservation and optimisation, our semantics depends on the optimisation which in turn depends on the flow analysis which in turn must be a conservative approximation of the semantics. In other words, the semantics, the transformation, and the analysis are all interdependent and we must therefore reason about them together.

The following section presents our variant of Featherweight Java. Section 3 presents the optimising semantics that does class-hierarchy analysis, speculative devirtualisation, and patching, and proves it type safe. Section 4 proves operational correctness, that is, the equivalence of the standard semantics and the optimising semantics.

2 The Language

We formalise our results in Featherweight Java [8] (FJ) extended with a static dispatch construct and facilities for dynamic class loading and dynamic patching, a language we call FJD. The language and its presentation follow the original FJ paper as closely as possible, and is modeled after our previous paper on type-safe method inlining [7]. In this extended abstract, we limit the informal explanations to static dispatch, dynamic class loading, and dynamic patching.

2.1 Syntax

The syntax of FJD is:

$$\begin{aligned}
P &::= (\overline{CD}; S; e) \\
CD &::= \text{class } C \text{ extends } C \{ \overline{C} \ \overline{f}; K \ \overline{M} \} \\
K &::= C(\overline{C} \ \overline{f}) \{ \text{super}(\overline{f}); \text{this}.\overline{f} = \overline{f}; \} \\
M &::= C \ m(\overline{C} \ \overline{x}) \{ \text{return } e; \} \\
e &::= x^\ell \mid e.f^\ell \mid \text{new}^\ell C(\overline{e}) \mid (C)^\ell e \mid e.m(\overline{e})^\ell \mid e.C::m(\overline{e})^\ell \mid \\
&\quad \text{dynnew}^\ell \mid \text{let } \overline{x} = \overline{e} \text{ in } x.C::m(\overline{y}) \text{ patchto}^\ell x.m(\overline{y})
\end{aligned}$$

We use S to range over sets of labels. Each S is called a *patch set*.

There are eight forms of expression: variables x^ℓ , field selection $e.f^\ell$, object constructors $\text{new}^\ell C(\overline{e})$, casts $(C)^\ell e$, dynamic method invocations $e.m(\overline{e})^\ell$, static method invocations $e.C::m(\overline{e})^\ell$, combined dynamic class loading and object creation dynnew^ℓ , and patching (see below). Static method invocation invokes C 's version of method m on object e , which should be in C or one of its subclasses. Dynamic class loading loads a new class and creates a new object of it.

Patching is a conditional construct: $\text{let } \overline{x} = \overline{e} \text{ in } x.C::m(\overline{y}) \text{ patchto}^\ell x.m(\overline{y})$ evaluates to the static dispatch $x.C::m(\overline{y}) \{ \overline{x} := \overline{e} \}$ unless the label ℓ is in the patch set S , in which case it evaluates to the dynamic dispatch $x.m(\overline{y}) \{ \overline{x} := \overline{e} \}$. Our transformation uses patching to optimise a dynamic dispatch to a static dispatch with the label initially not in the patch set. If class loading invalidates the optimisation, it adds the label into the patch set. The construct is formalised in Section 3.3. We use a specialised patch construct that is combined with a standard let construct to simplify our proofs. A more general construct has subtle order of evaluation issues, and the specialised construct is sufficient for our needs. We reserve patching for our transformation, and assume that it does not appear in the source program. In fact, there is no rule in the standard semantics for patching. Note also that patch sets have to do with the patching construct, and are not relevant for source programs. We include both in the syntax to have a single language, which simplifies the formalisation.

Metavariable ℓ ranges over a set of labels. Notice that there is a label associated with every expression; we assume that labels are unique in the source program. For a program P , $\text{labels}(P)$ denotes the set of labels used in P . To simplify the technical definitions later, all the field names and argument names must be distinct. Any well-typed program can easily be transformed to satisfy these conditions. Function lab maps an expression to its label.

Some auxiliary definitions that are used in the rest of the paper appear in Figure 1. Unlike the FJ paper, we do not make the list of class declarations global, but have them appear explicitly as parameters to functions, predicates, and rules. Function $\text{fields}(\overline{CD}, C)$ returns a list of C 's fields and their types; $\text{mtype}(\overline{CD}, C, m)$ returns the type of method m in class C , this type has the form $\overline{C} \rightarrow C$ where C is the return type and \overline{C} are the argument types; $\text{mbody}(\overline{CD}, C, m)$ returns the body of method m in class C , this has the form (\overline{x}, e) where e is the expression to evaluate and \overline{x} are the parameter names; $\text{impl}(\overline{CD}, C, m)$ returns the class from which class C inherits method

Field Lookup:

$$\overline{fields}(\overline{CD}, \text{Object}) = \bullet \quad (1)$$

$$\frac{\overline{CD}(C) = \text{class } C \text{ extends } D \{ \overline{C} \ \overline{f}; \ K \ \overline{M} \} \quad \overline{fields}(\overline{CD}, D) = \overline{D} \ \overline{g}}{\overline{fields}(\overline{CD}, C) = \overline{D} \ \overline{g}, \overline{C} \ \overline{f}} \quad (2)$$

Method Type Lookup:

$$\frac{\overline{CD}(C) = \text{class } C \text{ extends } D \{ \overline{C} \ \overline{f}; \ K \ \overline{M} \} \quad B_0 \ m(\overline{B} \ \overline{x}) \{ \text{return } e; \} \in \overline{M}}{mtype(\overline{CD}, C, m) = \overline{B} \rightarrow B_0} \quad (3)$$

$$\frac{\overline{CD}(C) = \text{class } C \text{ extends } D \{ \overline{C} \ \overline{f}; \ K \ \overline{M} \} \quad m \text{ not defined in } \overline{M}}{mtype(\overline{CD}, C, m) = mtype(\overline{CD}, D, m)} \quad (4)$$

Method Body Lookup:

$$\frac{\overline{CD}(C) = \text{class } C \text{ extends } D \{ \overline{C} \ \overline{f}; \ K \ \overline{M} \} \quad B_0 \ m(\overline{B} \ \overline{x}) \{ \text{return } e; \} \in \overline{M}}{mbody(\overline{CD}, C, m) = (\overline{x}, e)} \quad (5)$$

$$\frac{\overline{CD}(C) = \text{class } C \text{ extends } D \{ \overline{C} \ \overline{f}; \ K \ \overline{M} \} \quad m \text{ not defined in } \overline{M}}{mbody(\overline{CD}, C, m) = mbody(\overline{CD}, D, m)} \quad (6)$$

Class of Method Lookup:

$$\frac{\overline{CD}(C) = \text{class } C \text{ extends } D \{ \overline{C} \ \overline{f}; \ K \ \overline{M} \} \quad B_0 \ m(\overline{B} \ \overline{x}) \{ \text{return } e; \} \in \overline{M}}{impl(\overline{CD}, C, m) = C :: m} \quad (7)$$

$$\frac{\overline{CD}(C) = \text{class } C \text{ extends } D \{ \overline{C} \ \overline{f}; \ K \ \overline{M} \} \quad m \text{ not defined in } \overline{M}}{impl(\overline{CD}, C, m) = impl(\overline{CD}, D, m)} \quad (8)$$

Valid Method Overriding:

$$\frac{mtype(\overline{CD}, D, m) = \overline{D} \rightarrow D_0 \text{ implies } \overline{C} = \overline{D} \text{ and } C_0 = D_0}{override(\overline{CD}, D, m, \overline{C} \rightarrow C_0)} \quad (9)$$

Fig. 1. Auxiliary Definitions

m (this might be C itself if C declares m), this has the form $D :: m$ where D is the class. Predicate $override(\overline{CD}, D, m, \overline{C} \rightarrow C)$ is true when method m of type $\overline{C} \rightarrow C$ may be declared in a subclass of D . It checks that if D declares or inherits m then it has the same type, as required by Java's type system. The more general rule with contravariant argument types and covariant result types could be used, and the results of this paper would still hold (with minor changes to some of our rules).

2.2 Standard Operational Semantics

The language has two different operational semantics. The one presented in this section can be thought of as the reference semantics for the language—any optimisation should simulate this semantics. The other semantics models the devirtualisation optimisation we are trying to formalise. It is a semantics rather than just a transformation because of the need to transform and patch at dynamic loading time. Details and further motivation are given in Section 3.

The semantics is the same as for FJ and our previous paper [7], except for the two new constructs and labels on the reduction relation. A reduction is optionally labeled with a class definition and list of field initialiser expressions. This label indicates the class that was loaded and what initial field values were used. The sequence of such pairs is called the dynamic load sequence of the execution. The semantics appears in Figure 2. Metavariable X ranges over evaluation contexts, which are expressions with exactly one hole; $X(e)$ denotes the expression formed by replacing the hole in X by the expression e . Unlike the FJ paper, in addition to making the list of class declarations explicit in the rules we make the evaluation context explicit as well.

Because the language is functional and each class has exactly one constructor of a particular form, the values of the language, which are all objects, can be represented using object constructors

$$\frac{fields(\overline{CD}, C) = \overline{C} \ \overline{f}}{(\overline{CD}; S; X\langle new^{\ell_1} C(\overline{e}) . f_i^{\ell_2} \rangle) \mapsto_s (\overline{CD}; S; X\langle e_i \rangle)} \quad (10)$$

$$\frac{\overline{CD} \vdash C <: D}{(\overline{CD}; S; X\langle (D)^{\ell_1} new^{\ell_2} C(\overline{e}) \rangle) \mapsto_s (\overline{CD}; S; X\langle new^{\ell_2} C(\overline{e}) \rangle)} \quad (11)$$

$$\frac{mbody(\overline{CD}, C, m) = (\overline{x}, e_0)}{(\overline{CD}; S; X\langle new^{\ell_1} C(\overline{e}) . m(\overline{d})^{\ell_2} \rangle) \mapsto_s (\overline{CD}; S; X\langle e_0\{\text{this}, \overline{x} := new^{\ell_1} C(\overline{e}), \overline{d}\} \rangle)} \quad (12)$$

$$\frac{mbody(\overline{CD}, D, m) = (\overline{x}, e_0)}{(\overline{CD}; S; X\langle new^{\ell_1} C(\overline{e}) . D::m(\overline{d})^{\ell_2} \rangle) \mapsto_s (\overline{CD}; S; X\langle e_0\{\text{this}, \overline{x} := new^{\ell_1} C(\overline{e}), \overline{d}\} \rangle)} \quad (13)$$

$$\frac{CD = \text{class } C \text{ extends } \dots}{(\overline{CD}; S; X\langle dynnew^{\ell} \rangle) \xrightarrow{CD, \overline{e}}_s (\overline{CD}, CD; S; X\langle new C(\overline{e}) \rangle)} \quad (14)$$

Fig. 2. Standard Operational Semantics

$new^{\ell} C(\overline{e})$. Field access reduces to the appropriate element of \overline{e} . The cast $(D)^{\ell_1} new^{\ell_2} C(\overline{e})$ reduces to the object $new^{\ell_2} C(\overline{e})$ if C is a subclass of D . If C is not a subclass of D then the cast is irreducible representing that the cast fails as a checked run-time error. The reduction rule for method invocation $new^{\ell_1} C(\overline{e}) . m(\overline{d})^{\ell_2}$ looks up the method body of m in C , if this is $(\ell, \overline{x}, e_0)$ then the reduced expression is the body e_0 with the actuals \overline{d} substituted for the formals \overline{x} and the object $new^{\ell_1} C(\overline{e})$ substituted for **this**. Static method invocation $new^{\ell_1} C(\overline{e}) . D::m(\overline{d})^{\ell}$ reduces similarly except that the method is looked up in D , not C . Note that this method lookup can be done at compile time and a static method invocation can be implemented as a direct call rather than an indirect call through a virtual-dispatch table. The rule for $dynnew^{\ell}$ simply adds the new class to the list of classes in the program and reduces to a **new** expression using the field initialisers. Because patching is reserved for our transformation, the standard semantics does not refer to or change the patch set, nor does it have a rule for patching.

An irreducible expression is *stuck* if it is of the form x , $X\langle e.f^{\ell} \rangle$, $X\langle e.m(\overline{e})^{\ell} \rangle$, or $X\langle e.D::m(\overline{e})^{\ell} \rangle$ (or, for the optimising semantics, $X\langle \text{let } \overline{x} = \overline{e} \text{ in } x.C::m(\overline{y}) \text{ patchto}^{\ell} x.m(\overline{y}) \rangle$). The type system prevents stuck expressions from occurring during execution of a program. Irreducible expressions that are not stuck are of the form $v::new^{\ell} C(\overline{v})$ or $X\langle (C)^{\ell_1} new^{\ell_2} D(\overline{e}) \rangle$ where D is not a subclass of C ; the former represents normal termination with a fully evaluated object, the latter represents a failed cast.

2.3 Type System

The type system consists of the following judgements:

Judgement	Meaning
$\overline{CD} \vdash C <: D$	C is a subtype of D
$\overline{CD}; \Gamma \vdash e \in C$	e is well formed and of type C
$\overline{CD} \vdash M \text{ OK in } C$	M is well formed in class C
$\overline{CD} \vdash CD \text{ OK}$	CD is well formed
$\vdash P \in C$	P is well formed and of type C

A typing context Γ has the form $\overline{x}:\overline{C}$ where there are no duplicate variable names. The only types are the names of classes, and such a type includes all instances of that class and its subclasses. The rules appear in Figure 3. The bar notation denotes sequences of typing judgements, so $\overline{CD}; \Gamma \vdash \overline{e} \in \overline{C}$ abbreviates $\overline{CD}; \Gamma \vdash e_1 \in C_1, \dots, \overline{CD}; \Gamma \vdash e_n \in C_n$.

The rules for constructors and method invocation check that each actual has a subtype of the corresponding formal. The typing rule for dynamic method dispatch looks up the type of the method in the class of the receiver. The typing rule for static method dispatch $e.D::m(\overline{e})^{\ell}$ requires that e has some subtype of D and looks up the type of the method in D . Note that in the rule for **patchto**, the two branches must have the same type. While the first expression (for our

Subtyping:

$$\frac{}{\overline{CD} \vdash C <: \overline{C}} \quad \frac{\overline{CD} \vdash C <: D \quad \overline{CD} \vdash D <: E}{\overline{CD} \vdash C <: E} \quad \frac{\overline{CD}(C) = \text{class } C \text{ extends } D \{ \dots \}}{\overline{CD} \vdash C <: D} \quad (15)$$

Expression Typing:

$$\overline{CD}; \Gamma \vdash x^\ell \in \Gamma(x) \quad (16)$$

$$\frac{\overline{CD}; \Gamma \vdash e_0 \in C_0 \quad \text{fields}(\overline{CD}, C_0) = \overline{C} \ \overline{f}}{\overline{CD}; \Gamma \vdash e_0.f_i^\ell \in C_i} \quad (17)$$

$$\frac{\text{fields}(\overline{CD}, C) = \overline{D} \ \overline{f} \quad \overline{CD}; \Gamma \vdash \overline{e} \in \overline{E} \quad \overline{CD} \vdash \overline{E} <: \overline{D}}{\overline{CD}; \Gamma \vdash \text{new}^\ell C(\overline{e}) \in C} \quad (18)$$

$$\frac{\overline{CD}; \Gamma \vdash e_0 \in D}{\overline{CD}; \Gamma \vdash (C)^\ell e_0 \in C} \quad (19)$$

$$\frac{\overline{CD}; \Gamma \vdash e_0 \in C_0 \quad \text{mtype}(\overline{CD}, C_0, m) = \overline{D} \rightarrow C \quad \overline{CD}; \Gamma \vdash \overline{e} \in \overline{C} \quad \overline{CD} \vdash \overline{C} <: \overline{D}}{\overline{CD}; \Gamma \vdash e_0.m(\overline{e})^\ell \in C} \quad (20)$$

$$\frac{\overline{CD}; \Gamma \vdash e_0 \in C_0 \quad \overline{CD} \vdash C_0 <: D \quad \text{mtype}(\overline{CD}, D, m) = \overline{E} \rightarrow E_0 \quad \overline{CD}; \Gamma \vdash \overline{e} \in \overline{C} \quad \overline{CD} \vdash \overline{C} <: \overline{E}}{\overline{CD}; \Gamma \vdash e_0.D : m(\overline{e})^\ell \in E_0} \quad (21)$$

$$\overline{CD}; \Gamma \vdash \text{dynnew}^\ell \in \text{Object} \quad (22)$$

$$\frac{\overline{CD}; \Gamma \vdash \overline{e} \in \overline{C} \quad \overline{CD}; \Gamma, \overline{x} : \overline{C} \vdash x.C : m(\overline{y})^\ell \in D \quad \overline{CD}; \Gamma, \overline{x} : \overline{C} \vdash x.m(\overline{y})^\ell \in D}{\overline{CD}; \Gamma \vdash \text{let } \overline{x} = \overline{e} \text{ in } x.C : m(\overline{y}) \text{ patchto}^\ell x.m(\overline{y}) \in D} \quad (23)$$

Method Typing:

$$\frac{\overline{CD}; \text{this} : C, \overline{x} : \overline{C} \vdash e_0 \in E_0 \quad \overline{CD} \vdash E_0 <: C_0 \quad \overline{CD}(C) = \text{class } C \text{ extends } D \{ \dots \} \quad \text{override}(\overline{CD}, D, m, \overline{C} \rightarrow C_0)}{\overline{CD} \vdash C_0 \ m(\overline{C} \ \overline{x}) \{ \text{return } e_0; \} \text{ OK in } C} \quad (24)$$

Class Typing:

$$\frac{\overline{CD} \vdash \overline{M} \text{ OK in } C \quad \text{fields}(\overline{CD}, D) = \overline{D} \ \overline{g} \quad K = C(\overline{D} \ \overline{g}, \overline{C} \ \overline{f}) \{ \text{super}(\overline{g}); \text{this}.\overline{f} = \overline{f}; \}}{\overline{CD} \vdash \text{class } C \text{ extends } D \{ \overline{C} \ \overline{f}; K \ \overline{M} \} \text{ OK}} \quad (25)$$

Program Typing:

$$\frac{\overline{CD} \vdash \overline{CD} \text{ OK} \quad \overline{CD}; \bullet \vdash e \in C}{\vdash (\overline{CD}; S; e) \in C} \quad (26)$$

Fig. 3. Typing Rules

transformation) is a speculative optimisation that will be patched when it is no longer semantically correct, it continues to type check after class loading.

The rules are syntax directed, with the exception of the rules for subtyping. So, disregarding the details of how subtyping judgments are derived, for any program there is exactly one derivation possible. Thus for a program P and any ℓ appearing in it, each expression labeled by ℓ has a uniquely determined static type. Each ℓ labels at most one expression. Let $\text{static-type}(P, \ell)$ be the type of ℓ in program P .

In the presence of dynamic loading it is not enough that the original program type checks. All dynamically loaded classes must also type check for there to be a type safety guarantee. This is formalised as follows. An execution of a program with classes \overline{CD} with dynamic load sequence $\langle (\overline{CD}_1, \overline{e}_1), \dots \rangle$ (this sequence could be finite or infinite) is type correct exactly when

$$\vdash (\overline{CD}, \overline{CD}_1, \dots, \overline{CD}_i; \emptyset; \text{new}^\ell C(\overline{e}_i)) \in C$$

for all appropriate i where $\overline{CD}_i = \text{class } C \text{ extends } \dots$ and ℓ is fresh. Given these definitions, the following type soundness result holds. Its proof is similar to standard type soundness proofs [8].

Theorem 1. *A type correct execution of a well-typed program cannot get stuck.*

3 Devirtualisation

This section presents a devirtualisation optimisation that speculatively devirtualises given current information and then patches when that information is invalidated by dynamic class loading.

3.1 Class Hierarchy Analysis

Any devirtualisation optimisation is based on a flow analysis. This flow analysis maps each expression in the program to a set of classes such that the expression always evaluates to an instance of one of those classes. In practice, a just-in-time compiler can only afford to use a simple analysis such as Class Hierarchy Analysis (CHA) or a variant of it. CHA maps an expression to all the classes that are subtypes of the static type of the expression. This is a crude approximation to flow set, but surprisingly effective in practice.

Formally, $subclasses(P, C)$ is the set of subclasses of C (including C) in program P . CHA is defined as:

$$CHA(P, \ell) = subclasses(P, static_type(P, \ell))$$

3.2 Program Transformation

The program transformation uses CHA to transform a program fragment in a compositional fashion. It transforms each program fragment into a similar program fragment with the same label, and turns some dynamic method invocations into patchable static method invocations. Specifically, a dynamic call is changed to a static call when CHA determines that there is a unique target method.

The transformation consists of these parts, and appears in Figure 4.

Transformation	Meaning
$\llbracket CD \rrbracket_P$	the transformation of CD within program P
$\llbracket M \rrbracket_P$	the transformation of M within program P
$\llbracket e \rrbracket_P$	the transformation of e within program P
$\begin{aligned} \llbracket \text{class } C \text{ extends } D \{ \bar{C} \ \bar{f}; K \ \bar{M} \} \rrbracket_P &= \text{class } C \text{ extends } D \{ \bar{C} \ \bar{f}; K \ \llbracket \bar{M} \rrbracket_P \} \\ \llbracket D \ m(\bar{E} \ \bar{x}) \ \{ \text{return } e; \} \rrbracket_P &= D \ m(\bar{E} \ \bar{x}) \ \{ \text{return } \llbracket e \rrbracket_P; \} \end{aligned}$	
$\llbracket x^\ell \rrbracket_P$	$= x^\ell$
$\llbracket e.f^\ell \rrbracket_P$	$= \llbracket e \rrbracket_P.f^\ell$
$\llbracket \text{new}^\ell D(\bar{e}) \rrbracket_P$	$= \text{new}^\ell D(\llbracket \bar{e} \rrbracket_P)$
$\llbracket (D)^\ell e \rrbracket_P$	$= (D)^\ell \llbracket e \rrbracket_P$
$\llbracket e.m(\bar{e})^\ell \rrbracket_P$	$= \text{let } x, \bar{x} = \llbracket e \rrbracket_P, \llbracket \bar{e} \rrbracket_P \text{ in}$ $\quad x.D : : m(\bar{x}) \text{ patchto}^\ell x.m(\bar{x})$
$\text{where } \forall E \in CHA(P, lab(e)) : impl(classes(P), E, m) = D : : m, \text{ and } x \text{ and } \bar{x} \text{ are fresh}$	
$\llbracket e.m(\bar{e})^\ell \rrbracket_P$	$= \llbracket e \rrbracket_P.m(\llbracket \bar{e} \rrbracket_P)^\ell$
otherwise	
$\llbracket e.D : : m(\bar{e})^\ell \rrbracket_P$	$= \llbracket e \rrbracket_P.D : : m(\llbracket \bar{e} \rrbracket_P)^\ell$
$\llbracket \text{dynnew}^\ell \rrbracket_P$	$= \text{dynnew}^\ell$

Fig. 4. The Transformation of Dynamic to Static Dispatch

The transformation preserves typability, the proof is straightforward.

Theorem 2. *If $P = (\overline{CD}; S; d)$, and C , M , and e appear in P then:*

- *If $\overline{CD} \vdash CD$ OK then $\llbracket \overline{CD} \rrbracket_P \vdash \llbracket CD \rrbracket_P$ OK.*
- *If $\overline{CD} \vdash M$ OK in D then $\llbracket \overline{CD} \rrbracket_P \vdash \llbracket M \rrbracket_P$ OK in D .*
- *If $\overline{CD}; \Gamma \vdash e \in D$ then $\llbracket \overline{CD} \rrbracket_P; \Gamma \vdash \llbracket e \rrbracket_P \in D$.*

3.3 Optimised Semantics

The optimised semantics uses CHA to speculatively devirtualise method calls, and patching to undo these optimisations when dynamically loaded classes invalidate the speculative assumption. The initial program is optimised, so is any dynamically loaded class and its corresponding field initialisers. The existing classes could also be reoptimised at dynamic load time, but we do not include this step (it would be straightforward to add).

Usually an operational semantics is defined just by a reduction relation between program states. For FJD however, the optimised semantics is defined by its initial states and by a reduction relation. The initial execution state for a program $P = (\overline{CD}; S; e)$ is $(\llbracket \overline{CD} \rrbracket_P; S; \llbracket e \rrbracket_P)$. The reduction relation is defined in Figure 5.

$$\frac{fields(\overline{CD}, C) = \overline{C} \ \overline{f}}{(\overline{CD}; S; X\langle new^{\ell_1} C(\overline{e}) . f_i^{\ell_2} \rangle) \mapsto_o (\overline{CD}; S; X\langle e_i \rangle)} \quad (27)$$

$$\frac{\overline{CD} \vdash C <: D}{(\overline{CD}; S; X\langle (D)^{\ell_1} new^{\ell_2} C(\overline{e}) \rangle) \mapsto_o (\overline{CD}; S; X\langle new^{\ell_2} C(\overline{e}) \rangle)} \quad (28)$$

$$\frac{mbody(\overline{CD}, C, m) = (\overline{x}, e_0)}{(\overline{CD}; S; X\langle new^{\ell_1} C(\overline{e}) . m(\overline{d})^{\ell_2} \rangle) \mapsto_o (\overline{CD}; S; X\langle e_0 \{ \text{this}, \overline{x} := new^{\ell_1} C(\overline{e}), \overline{d} \} \rangle)} \quad (29)$$

$$\frac{mbody(\overline{CD}, D, m) = (\overline{x}, e_0)}{(\overline{CD}; S; X\langle new^{\ell_1} C(\overline{e}) . D :: m(\overline{d})^{\ell_2} \rangle) \mapsto_o (\overline{CD}; S; X\langle e_0 \{ \text{this}, \overline{x} := new^{\ell_1} C(\overline{e}), \overline{d} \} \rangle)} \quad (30)$$

$$\frac{CD = \text{class } C \text{ extends } \dots}{P = (\overline{CD}, CD; S; X\langle new^{\ell} C(\overline{e}) \rangle) \quad CD' = \llbracket CD \rrbracket_P \quad \overline{e}' = \llbracket \overline{e} \rrbracket_P \quad S' = S \cup poly(P)} \quad (31)$$

$$\frac{(\overline{CD}; S; X\langle dynnew^{\ell} \rangle) \xrightarrow{CD, \overline{e}} (\overline{CD}, CD'; S'; X\langle new^{\ell} C(\overline{e}') \rangle) \quad \ell \notin S \quad x\{\overline{x} := \overline{e}\} = new^{\ell_1} C(\overline{d}_1) \quad \overline{y}\{\overline{x} := \overline{e}\} = \overline{d}_2 \quad mbody(\overline{CD}, D, m) = (\overline{z}, e_0)}{(\overline{CD}; S; X\langle e_0 \{ \text{this}, \overline{z} := new^{\ell_1} C(\overline{d}_1), \overline{d}_2 \} \rangle) \xrightarrow{L} P} \quad (32)$$

$$\frac{(\overline{CD}; S; X\langle \text{let } \overline{x} = \overline{e} \text{ in } x.D :: m(\overline{y}) \text{ patchto}^{\ell} x.m(\overline{y}) \rangle) \xrightarrow{L} P \quad \ell \in S \quad x\{\overline{x} := \overline{e}\} = new^{\ell_1} C(\overline{d}_1) \quad \overline{y}\{\overline{x} := \overline{e}\} = \overline{d}_2 \quad mbody(\overline{CD}, C, m) = (\overline{z}, e_0)}{(\overline{CD}; S; X\langle e_0 \{ \text{this}, \overline{z} := new^{\ell_1} C(\overline{d}_1), \overline{d}_2 \} \rangle) \xrightarrow{L} P} \quad (33)$$

Fig. 5. Optimised Operational Semantics

The main differences between the semantics are the rule for dynamic new and patching. In the optimised semantics the newly loaded class and field initialisers are optimised using the new program. Furthermore, the patch set is updated to include all labels that are now polymorphic call sites—the set $poly(P)$, which is defined as follows.

$$poly(P) = \{ \ell \mid \text{let } \dots, x=e, \dots \text{ in } x.D :: m(\overline{x}) \text{ patchto}^{\ell} \dots \text{ in } P \text{ and } \neg \forall E \in CHA(P, lab(e)) : impl(classes(P), E, m) = D :: m \}$$

The patching construct executes the static dispatch if the label is not in the patch set, and executes the dynamic dispatch if the label is in the patch set. It is zero time, in that, it does not reduce to the dispatch but rather to what the dispatch reduces to. This is a technical device to simplify the proof of operational correctness. It also reflects the cheap implementation cost of the construct. Other than these remarks, the two rules just combine standard rules for let, the dispatch rules, and the condition.

The optimised semantics is also type safe. The proof is straightforward given the previous theorem.

Theorem 3. *A type correct, optimised execution of a well-typed program cannot get stuck.*

4 Correctness

This section will prove that the optimisation is correct. Specifically it will show that the optimised semantics simulates the standard semantics and vice versa.

To state the result we need a *correspondence* relation. This relation generalises the transformation slightly to reflect the fact that the transformation is applied at consecutive loading points rather than all at once. Its definition appears in Figure 6. Essentially, where the left program has a dynamic dispatch the right program may have one of two expressions. It can have a corresponding dynamic dispatch. It can also have a static dispatch that is patched to a dynamic dispatch if the dynamic dispatch is monomorphic in the current program (the subscript P on the relation) or if the patch label is in the current patch set (the subscript S on the relation).

$$\overline{\text{corresponds}_{P,S}(\mathbf{x}^\ell, \mathbf{x}^\ell)} \quad (34)$$

$$\frac{\text{corresponds}_{P,S}(\mathbf{e}_1, \mathbf{e}_2)}{\text{corresponds}_{P,S}(\mathbf{e}_1.\mathbf{f}^\ell, \mathbf{e}_2.\mathbf{f}^\ell)} \quad (35)$$

$$\frac{\text{corresponds}_{P,S}(\bar{\mathbf{e}}_1, \bar{\mathbf{e}}_2)}{\text{corresponds}_{P,S}(\text{new}^\ell C(\bar{\mathbf{e}}_1), \text{new}^\ell C(\bar{\mathbf{e}}_2))} \quad (36)$$

$$\frac{\text{corresponds}_{P,S}(\mathbf{e}_1, \mathbf{e}_2)}{\text{corresponds}_{P,S}((C)^\ell \mathbf{e}_1, (C)^\ell \mathbf{e}_2)} \quad (37)$$

$$\frac{\text{corresponds}_{P,S}(\mathbf{e}_1, \mathbf{e}_2) \quad \text{corresponds}_{P,S}(\bar{\mathbf{e}}_1, \bar{\mathbf{e}}_2)}{\text{corresponds}_{P,S}(\mathbf{e}_1.m(\bar{\mathbf{e}}_1)^\ell, \mathbf{e}_2.m(\bar{\mathbf{e}}_2)^\ell)} \quad (38)$$

$$\frac{\begin{array}{l} \forall D \in \text{CHA}(P, \text{lab}(\mathbf{e}_1)) : \text{impl}(\text{classes}(P), D, m) = C : m \\ \text{corresponds}_{P,S}(\mathbf{e}_1, \mathbf{e}_2) \quad \text{corresponds}_{P,S}(\bar{\mathbf{e}}_1, \bar{\mathbf{e}}_2) \\ \mathbf{e}' = \text{let } \mathbf{x}, \bar{\mathbf{x}} = \mathbf{e}_2, \bar{\mathbf{e}}_2 \text{ in } \mathbf{x}.C : m(\bar{\mathbf{x}}) \text{ patchto}^\ell \mathbf{x}.m(\bar{\mathbf{x}}) \end{array}}{\text{corresponds}_{P,S}(\mathbf{e}_1.m(\bar{\mathbf{e}}_1)^\ell, \mathbf{e}')} \quad (39)$$

$$\frac{\begin{array}{l} \ell \in S \quad \text{corresponds}_{P,S}(\mathbf{e}_1, \mathbf{e}_2) \quad \text{corresponds}_{P,S}(\bar{\mathbf{e}}_1, \bar{\mathbf{e}}_2) \\ \mathbf{e}' = \text{let } \mathbf{x}, \bar{\mathbf{x}} = \mathbf{e}_2, \bar{\mathbf{e}}_2 \text{ in } \mathbf{x}.C : m(\bar{\mathbf{x}}) \text{ patchto}^\ell \mathbf{x}.m(\bar{\mathbf{x}}) \end{array}}{\text{corresponds}_{P,S}(\mathbf{e}_1.m(\bar{\mathbf{e}}_1)^\ell, \mathbf{e}')} \quad (40)$$

$$\frac{\text{corresponds}_{P,S}(\mathbf{e}_1, \mathbf{e}_2) \quad \text{corresponds}_{P,S}(\bar{\mathbf{e}}_1, \bar{\mathbf{e}}_2)}{\text{corresponds}_{P,S}(\mathbf{e}_1.C : m(\bar{\mathbf{e}}_1)^\ell, \mathbf{e}_2.C : m(\bar{\mathbf{e}}_2)^\ell)} \quad (41)$$

$$\overline{\text{corresponds}_{P,S}(\text{dynnew}^\ell, \text{dynnew}^\ell)} \quad (42)$$

$$\frac{\text{corresponds}_{P,S}(\mathbf{e}_1, \mathbf{e}_2)}{\text{corresponds}_{P,S}(D \ m(\bar{C} \ \bar{\mathbf{x}}) \ \{\text{return } \mathbf{e}_1; \}, D \ m(\bar{C} \ \bar{\mathbf{x}}) \ \{\text{return } \mathbf{e}_2; \})} \quad (43)$$

$$\frac{\text{corresponds}_{P,S}(\bar{M}_1, \bar{M}_2)}{\text{corresponds}_{P,S}(\text{class } C \text{ extends } D \ \{\bar{C} \ \bar{\mathbf{f}}; K \ \bar{M}_1\}, \text{class } C \text{ extends } D \ \{\bar{C} \ \bar{\mathbf{f}}; K \ \bar{M}_2\})} \quad (44)$$

$$\frac{P = (\bar{CD}_1; S_1; \mathbf{e}_1) \quad \text{corresponds}_{P,S_2}(\bar{CD}_1, \bar{CD}_2) \quad \text{corresponds}_{P,S_2}(\mathbf{e}_1, \mathbf{e}_2)}{\text{corresponds}(P, (\bar{CD}_1; S_2; \mathbf{e}_2))} \quad (45)$$

Fig. 6. The Correspondence Relation

The transformation is a special case of correspondence.

Theorem 4.

- If P' is the initial state for program P in the optimised semantics then $\text{corresponds}(P, P')$.
- For any P and S , $\text{corresponds}_{P,S}(\text{CD}, \llbracket \text{CD} \rrbracket_P)$.
- For any P and S , $\text{corresponds}_{P,S}(e, \llbracket e \rrbracket_P)$.

The optimised semantics simulates the standard semantics and vice versa.

Theorem 5. If $\text{corresponds}(P_1, P'_1)$ then:

- If $P_1 \xrightarrow{L}_S P_2$ then $P'_1 \xrightarrow{L}_O P'_2$ and $\text{corresponds}(P_2, P'_2)$ for some P'_2 .
- If $P'_1 \xrightarrow{L}_O P'_2$ then $P_1 \xrightarrow{L}_S P_2$ and $\text{corresponds}(P_2, P'_2)$ for some P_2 .

We have proven both theorems, but omit the proofs due to a lack of space. The proofs appear in a full version of the paper, which will be available from our websites.

References

1. Matthew Arnold and Barbara Ryder. Thin guards: A simple and effective technique for reducing the penalty of dynamic class loading. In *Sixteenth European Conference on Object-Oriented Programming*, volume 2374 of *Lecture Notes in Computer Science*, pages 498–524, Malaga Spain, June 2002.
2. Michal Cierniak, Marsha Eng, Neal Glew, Brian Lewis, and James Stichnoth. The open runtime platform: A flexible high-performance managed runtime environment. *Intel Technical Journal*, 7(1), February 2003.
3. Michal Cierniak, Guei-Yuan Lueh, and James Stichnoth. Practicing judo: Java under dynamic optimizations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–26, British Columbia, Canada, June 2000.
4. Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of European Conference on Object-Oriented Programming*, pages 77–101, Aarhus, Denmark, August 1995. Springer-Verlag (LNCS 952).
5. David Detlefs and Ole Agesen. Inlining of virtual methods. In *Proceedings of ECOOP'99, European Conference on Object-Oriented Programming*, pages 258–278. Springer-Verlag (LNCS 1628), 1999.
6. Neal Glew. An efficient class and object encoding. In *Proceedings of OOPSLA'00, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 311–324, Minneapolis, Minnesota, October 2000.
7. Neal Glew and Jens Palsberg. Type-safe method inlining. In *Proceedings of ECOOP'02, European Conference on Object-Oriented Programming*, pages 525–544. Springer-Verlag (LNCS 2374), Malaga, Spain, June 2002.
8. Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 132–146, Denver, CO, USA, October 1999.
9. Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Hideaki Komatsu, and Toshio Nakatani. A study of devirtualization techniques for a Java just-in-time compiler. In *Proceedings of the Fifteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*, 2000.
10. Greg Morrisett, David Tarditi, Perry Cheng, Christopher Stone, Robert Harper, and Peter Lee. The TIL/ML compiler: Performance and safety through types. In *ACM SIGPLAN Workshop on Compiler Support for System Software*, Tucson, AZ, USA, February 1996.
11. Greg Morrisett, David Walker, Karl Cray, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.
12. Vugranam Sreedhar, Michael Burke, and Jong-Deok Choi. A framework for interprocedural optimization in the presence of dynamic class loading. In *Proceedings of PLDI'00, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–207, 2000.
13. David Tarditi, Greg Morrisett, Perry Cheng, Christopher Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, PA, USA, May 1996. ACM Press.
14. Andrew Wright, Suresh Jagannathan, Cristian Ungureanu, and Aaron Hertzmann. Compiling Java to a typed lambda-calculus: A preliminary report. In *ACM Workshop on Types in Compilation*, Kyoto, Japan, March 1998.