Type-Safe Optimisation of Plugin Architectures

Neal Glew[1], Jens Palsberg[2], and Christian Grothoff[3]

[1] Intel Corporation, Santa Clara, CA 95054, USA
aglew@acm.org
[2] UCLA Computer Science Dept, Los Angeles, CA 90095, USA
palsberg@ucla.edu
[3] Purdue University, Dept. of Computer Science, West Lafayette, IN 47907, USA
christian@grothoff.org

Abstract. Programmers increasingly implement plugin architectures in type-safe object-oriented languages such as Java. A virtual machine can dynamically load class files containing plugins, and a JIT compiler can do optimisations such as method inlining. Until now, the best known approach to type-safe method inlining in the presence of dynamic class loading is based on Class Hierarchy Analysis. Flow analyses that are more powerful than Class Hierarchy Analysis lead to more inlining but are more time consuming and not known to be type safe. In this paper we present and justify a new approach to type-safe method inlining in the presence of dynamic class loading. First we present experimental results that show that there are major advantages to analysing all locally available plugins at start-up time. If we analyse the locally available plugins at start-up time, then flow analysis is only needed at start-up time and when downloading plugins from the Internet, that is, when long pauses are expected anyway. Second, inspired by the experimental results, we design a new framework for type-safe method inlining which is based on a new type system and an existing flow analysis. In the new type system, a type is a pair of Java types, one from the original program and one that reflects the flow analysis. We prove that method inlining preserves typability, and the experimental results show that the new approach inlines considerably more call sites than Class Hierarchy Analysis.

1 Introduction

In a rapidly changing world, software has a better chance of success when it is extensible. Rather than having a fixed set of features, extensible software allows new features to be added on the fly. For example, modern browsers such as Firefox, Konqueror, Mozilla, and Viola [25] allow downloading of plug-ins that enable the browser to display new types of content. Using plugins can also help keep the core of the software smaller and make large projects more manageable thanks to the resulting modularisation. Plugin architectures have become a common approach to achieving extensibility and include well-known software such as Eclipse and Jedit.

While good news for users, plug-ins architectures are challenging for optimising compilers. This paper investigates the optimisation of software that has a plug-in architecture and that is written in a type-safe object-oriented language. Our focus is on method inlining, one of the most important and most studied optimisations for object-oriented languages.

Consider the following typical snippet of Java code for loading and running a plugin.

```
String className = ...;
Class c = Class.forName(className);
Object o = c.newInstance();
Runnable p = (Runnable) o;
p.run();
```

The first line gets from somewhere the name of a plugin class. The list of plugins is typically supplied in the system configuration and loaded using I/O, preventing the compiler from doing a data-flow analysis to determine all possible plugins. The second line loads a plugin class with the given name. The third line creates an instance of the plugin class, which is subsequently cast to an interface and used.

In the presence of this dynamic loading, a compiler has two choices: either treat dynamic-loading points very conservatively or make speculative optimisations based on currently loaded classes only. The former can pollute the analysis of much of the program, potentially leading to little optimisation. The latter can potentially lead to more optimisation, but dynamically-loaded code might *invalidate* earlier optimisation decisions, and thus require the compiler to undo the optimisations. When a method inlining is invalidated by class loading, the run-time must *revirtualise* the call, that is, replace the inlined code with a virtual call. The observation that invalidations can happen easily in a system that uses plugins leads to the question:

> **Question:** If an optimising compiler for a plug-in architecture inlines aggressively, will it have to revirtualise frequently?

This paper presents experimental results for Eclipse and Jedit that quantify the potential invalidations and suggest how to significantly decrease the number of invalidations. We count which sites are likely candidates for future invalidation, which sites are unlikely to require invalidation, and which sites are guaranteed to stay inlined forever. These numbers suggest that speculative optimisation is beneficial and that invalidation can be kept manageable.

In addition to the goal of inlining more and revirtualising less, we want method inlining to preserve typability. This paper shows how to do inlining and revirtualisation in a way that preserves typability of the intermediate representation. The quest for preserving typability stems from the success of several compilers that use typed intermediate languages [9,15,16,17,26] to give debugging and optimisation benefits [16,24]. A bug in a compiler that discards type information might result in a run-time error, such as a segmentation violation, that should be impossible in a typed language. On the other hand, if optimisations are type preserving, bugs can be found automatically by verifying that the compiler generates an intermediate respresentation that type checks. Additionally, preserving the types in the intermediate code may help guide other optimisations. So it is desirable to write optimisations so that they preserve typability.

Most of the compilers that use typed intermediate languages are "ahead-of-time" compilers. Similar benefits are desired for "just-in-time" (JIT) compilers. A step towards that goal was taken by the Jikes Research Virtual Machine [1] for Java, whose JIT compilers preserve and exploit Java's static types in the intermediate representations, chiefly for optimisation purposes. However, those intermediate representations are not typed in the usual sense—there is no type checker that guarantees type soundness (David Grove, personal communication, 2004). In two previous papers we presented algorithms for type-safe method inlining. The first paper [11] handles a setting *without* dynamic class loading, and the second paper [10] handles a setting *with* dynamic class loading, but with the least-precise flow analysis possible (CHA). In this paper we improve significantly on the second paper by presenting a new transformation and type system that together can handle a similar class of flow analyses as in the first paper.

Our Results. We make two contributions. Our first contribution is to present experimental numbers for inlining and invalidation. These numbers show that if a compiler analyses all plugins that are locally available, then dynamically loading from these plugins will lead to a miniscule number of invalidations. In contrast, when dynamically loading an unanalysed plugin, the run-time will have to consider a significantly larger number of invalidations. In order to ensure that loading unanalzed plugins happens less frequently, the compiler should analyse all of the local plugins using the most powerful technique available. That observation motivates our second contribution, which is a new framework for type-safe method inlining. The new framework handles dynamic class loading and a wide range of flow analyses. The main technical innovation is a technique for changing type annotations both at speculative devirtualisation time and at revirtualisation time, solving the key issue that we identified but side stepped in our previous paper [10]. As in both our previous papers, we prove a formalisation of the optimisation correct and type preserving. Using the most-precise flow analysis in the permitted class, our new framework achieves precision comparable to 0-CFA [18,21].

2 An Experiment

Using the plugin architectures Eclipse and Jedit as our benchmark, we have conducted an experiment that addresses the following questions:

– How many call sites can be inlined?
– How many inlinings remain valid and how many can be invalidated?

– How much can be gained by preanalysing the plugins that are statically available?

Preanalysing plugins can be beneficial. Consider the code in Figure 1. The analysis can see that the plugin calls method `m` in `Main` and passes it an `Main.B2`; since `main` also calls `m` with a `Main.B1`, it is probably not a good idea to inline the `a.n()` call in `m` as it will be invalidated by loading the plugin. The analysis can also see which methods are overridden by the plugin, in this case only `run` of `Runnable` is. The analysis must still be conservative in some places, for example at the instantiation inside of the `for` loop, as this statement could load any plugin. But the analysis can gather much more information about the program and make decisions based on likely invalidations by dynamically loading the known plugins.

Being able to apply the inlining optimisation in the first place still depends on the flow analysis being powerful enough to establish the unique target. Thus, the answer to each of the three questions depends on the static analysis that is used to determine which call sites have a unique target. We have experimented with four different interprocedural flow analyses, all implemented for Java bytecode, here listed in order of increasing precision (the first three support type preservation, the last one does not):

– Class Hierarchy Analysis (CHA, [7,8])
– Rapid Type Analysis (RTA, [2,3])
– subset-based, context-insensitive, flow-insensitive flow analysis for type-preserving method inlining (TSMI, [11]) and
– subset-based, context-insensitive, flow-insensitive flow analysis (0-CFA, [18,21]).

In order to show that deoptimisation is a necessity for optimising compilers for plugin architectures, we also give the results for a simple intraprocedural flow analysis ("local") which corresponds to the number of inlinings that will never have to be deoptimised, even if arbitrary new code is added to the system. The "local" analysis essentially makes conservative assumptions about all arguments, including the possibility of being passed new types that are not known to the analysis. A run-time system that cannot perform deoptimisation is limited to the optimisations found by "local" if loading arbitrary plugins is to be allowed.

The implementations of the five analyses share as much code as possible; our goal was to create the fairest comparison, not to optimise the analysis time. All of our experiments were run with at most 1.8 GB of memory. (1.8 GB is the maximum total process memory for the Hotspot Java Virtual Machine running on OS X as reported by `top` and also the memory limit specified at the command line using the `-Xmx` option.)

We use two benchmarks in our experiments:

Jedit 4.2pre13. A free programmer's text editor which can be extended with plugins from `http://jedit.org/`, 865 classes; analysed with GNU classpath 0.09, from `http://www.classpath.org`, 2706 classes.

```
class Main {
  static Main main;
  public static void main(String[] args) throws Exception {
    main = new Main();
    for (int i=0;i<args.length;i++) {
      Class c = Class.forName(args[i]);
      Runnable p = (Runnable) c.newInstance();
      p.run(); // virtual if loaded plugins define multiple run methods
    }
    main.m(new B1()); // can stay optimised for given Plugin
  }
  void m(A a) { a.n(); // needs to be virtual for given Plugin }
  static abstract class A {
    abstract void n();
  }
  static class B1 extends A {
    void n() { }
  }
  static class B2 extends Main.A {
     void n() { }
  }
}
class Plugin implements Runnable {
  public void run() { new Main().m(new Main.B2()); }
}
```

Fig. 1. Example code loading a known plugin. The Plugin does not modify `Main.main`, which ensures that the call to `main.m()` can remain inlined. If only `Plugin` is loaded, `p.run()` can also be inlined. Pre-analysing `Plugin` reveals that `a.n()` should be virtual, even if the flow analysis of the code without `Plugin` may say otherwise.

Eclipse 3.0.1. An open extensible Integrated Development Environment from `http://www.eclipse.org/`, 22858 classes from the platform and the CDT, JDT, PDE and SDK components; analysed with Sun JDK 1.4.2 for Linux, 10277 classes (using the JARs dnsns, rt, sunrsasign, jsse, jce, charsets, sunjce_provider, ldapsec and localedata).

While we have "only" two benchmarks, note that the combined size of SPECjvm98 and SPECjbb2000 is merely 11% of the size of Eclipse. Furthermore, these are the only freely available large Java systems with plugin architectures that we are aware of. Analysing benchmarks, such as the SPEC benchmarks, that do not have plugins is pointless. We are not aware of any previously published results on 0-CFA for benchmarks of this size.

We will use *app* to denote the core application together with the plugins that are available for ahead-of-time analysis. Automatically drawing a clear line between plugins and the main application is difficult considering that parts of the "core" may only be reachable from certain plugins.

Usually, flow analyses are implemented with a form of reachability built in, and more powerful powerful analyses are better at reachability. To further ensure

Jedit	Can be inlined						Cannot be inlined		Total	
	Remain valid		Can be invalidated							
			By DLCW		By DLOW not DLCW					
	app	lib	app	lib	app	lib	app	lib	app	lib
Local	682	297	0	0	0	0	20252	7808	20934	8105
CHA	682	297	69	7	18720	6178	1463	1623	20934	8105
RTA	682	297	97	51	18723	6178	1432	1579	20934	8105
TSMI	682	297	99	59	19449	7091	704	658	20934	8105
0-CFA	682	297	103	83	19592	7191	557	534	20934	8105

Eclipse	Can be inlined						Cannot be inlined		Total	
	Remain valid		Can be invalidated							
			By DLCW		By DLOW not DLCW					
	app	lib	app	lib	app	lib	app	lib	app	lib
Local	15497	472	0	0	0	0	481939	26512	497436	26984
CHA	15497	472	4105	61	366114	20796	111720	5655	497436	26984
RTA	15497	472	9024	169	366169	20797	106746	5546	497436	26984
TSMI	15497	472	11479	439	420029	23097	50431	2976	497436	26984
0-CFA	15497	472	9921	46	428944	23971	43074	2495	497436	26984

Fig. 2. Experimental results; each number is a count of virtual call sites

a fair comparison of the analyses, reachability is first done once in the same way for all analyses. Then each of the analyses is run with reachability disabled. The initial reachability analysis is based on RTA and assumes that all of *app* is live, in particular, all local plugins are treated as roots for reachability. The analysis determines the part of the library (classpath, JDK) which is live, denoted *lib*, and then we remove the remainder of the library.

The combination *app + lib* is the “closed world” that is available to the ahead-of-time compiler, in contrast to all of the code that could theoretically be dynamically loaded from the “open world”. We use the abbreviations:

DLCW = Dynamic Loading from Closed World
DLOW = Dynamic Loading from Open World.

In other words, DLCW means loading a local plugin, whereas DLOW means loading a plugin from, say, the Internet.

Figure 2 shows the static number of virtual call sites that can be inlined under the respective circumstances. The numbers show that loading from the local set of plugins results in an extremely small number of possible invalidations (DLCW). The numbers also show that preanalyzing plugins is about 50% more effective for 0-CFA than for CHA: the number of additional devirtualisations is respectively 57% and 49% higher for 0-CFA after compensating for the higher number of devirtualisations of 0-CFA. When loading arbitrary code from the open world (DLOW), the compiler has to consider almost all devirtualised call sites for invalidation. Only a tiny fraction of all virtual calls can be guaranteed to never require revirtualisation in a setting with dynamic loading—a compiler that cannot revirtualise calls can only perform a fraction of the possible inlining optimisations.

The data also shows that TSMI and 0-CFA are quite close in terms of precision, which is good news since this means it is possible to use the type-safe variant without loosing many opportunities for optimisation. As expected, using 0-CFA or TSMI instead of CHA or RTA cuts in half the number of virtual calls left in the code after optimisation. Notice that for Eclipse, in the column for call sites that can be inlined and invalidated by DLCW, 0-CFA has a *smaller* number than TSMI. This is not an anomaly; on the contrary, it shows that 0-CFA is so good that it both identifies 7357 more call sites in *app* for inlining than TSMI *and* determines that many call sites cannot be invalidated by DLCW.

The closest related work to our experiment is the extant analysis of Sreedhar, Burke, and Choi [22] which determines whether a variable can only contain objects of classes from the closed world. They did not consider the more detailed question of whether inlining can be invalidated due to DLCW or only due to DLOW. Their largest benchmark was *jess* which has 112 classes.

3 Overview of Our Framework

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.

A New Type System. In later sections we will prove that TSMI supports type-safe method inlining for a setting with dynamic class loading. We use a new type system for the intermediate representation: each type is a pair of Java types. In this section we explain the main problem that lead us to the new type system. Our running example is an extended version of one from our paper on TSMI [11].

```
class B {
  B m() { return this; }
}

class C extends B {
  C f;
  B m() {
    return this.f;
  }
}
```

```
// code snippet 1:
B x = new C(); // x is a field
x = x.m();
x = ((B)new C()).m();

// code snippet 2:
B y;           // y is a field
if (...) { y = new C(); }
   else  { y = (B)dynnew; }
y = y.m();
```

The two code snippets contain three method calls, each to a receiver object of type B. CHA will for each method call determine that there are two possible target methods, namely B.m and C.m, so CHA will lead to inlining of *none* of the three call sites.

In snippet 1, which does not have dynamic loading, both of the calls have unique targets that are small code fragments, so it makes sense to inline these calls:

```
x = x.f;                        // does not type check
x = ((B)new C()).f              // does not type check
```

These two assignments do not type check because while this in class C has static type C, both x and (B)new C() have static type B. Since B has no f field, both field selections fail the type checker. As explained in our previous paper [11], we remedy this problem by changing static type information to reflect the more accurate information the flow analysis has. In particular, the flow analysis has determined that x and the cast expression only evaluate to objects of type C, and so we transform the static type information to produce the following well-typed code snippet:

```
C x = new C();
x = x.f;                        // type checks
x = ((C)new C()).f;             // type checks
```

To understand the problems introduced by dynamic class loading, let us consider code snippet 2. The method call y.m() has a unique target method *at least until the next dynamic class loading*. So it makes sense to inline the call, even though that decision may be invalidated later. To see how this may be achieved, the key question is:

Question: What is the flow set for dynnew ?

With CHA, the answer is given by the static type of dynnew, which is Object, and so the flow set is "all classes in the program". Since dynnew has no impact on the execution *until the next dynamic class loading*, we could assign dynnew the empty flow set! We extend TSMI to dynamic loading in this way. However, this idea runs into a difficulty quickly, as we explain next.

For code snippet 2, our previous approach transforms the types in a way that preserves well-typedness:

```
C y;                        // the type of y is changed to C
if (...) { y = new C(); }
   else  { y = (C)dynnew; } // the type cast is changed to C
y = y.m();
```

Let us now suppose that control reaches dynnew and that it loads and instantiates a class D which extends class B and is otherwise unrelated to class C. In the original code snippet 2, the cast of dynnew is to B, so it succeeds. However, in the transformed code snippet, the cast of dynnew is to C, so it fails. Thus, if we transform the types in the style of our previous paper [11] and we do not transform the types *again* at the time of evaluating dynnew, we change the meaning of the program!

The source of the difficulty is that a type cast can viewed as doing double duty: it does a run-time check and it helps the type checker. Our solution is to change the cast into a form that uses a *pair* of types. In code snippet 2, we would change the cast of dynnew to (B,C)dynnew. We say that B is the *original* type and that C is the *current* type. The current type is based on the flow analysis. The original type is used to do the run-time check while the current type is used to help the type checker. In fact, we need to change the entire type system and use pairs of types everywhere, not just in casts. Note, to be sound, the current type must be a subtype of the original type.

Armed with the idea of using pairs of types, we can now state the type of dynnew. The original type continues to be Object and the current type is derived from the flow set which is the empty set. The empty set corresponds to a type which is a subtype of all other types. To reflect that, we introduce a type Null and give dynnew the type (Object, Null). This has the pleasant side effect that we can remove an artificial requirement from the original formulation of TSMI, namely that all flow sets have to be nonempty.

Returning to code snippet 2, our approach will first transform the snippet into:

```
(B,C) y;                          // the type of y is changed to (B,C)
if (...) { y = new C(); }
   else  { y = (B,C)dynnew; }  // the type cast is changed to (B,C)
y = y.m();
```

Next, evaluating dynnew and thereby loading and instantiating a class D can be modeled as replacing dynnew with new D() as well as a new flow analysis of the program. The new analysis changes the current types, resulting in the following type-correct code:

```
(B,B) y;
if (...) { y = new C(); }
   else  { y = (B,D)new D(); }
y = y.m();
```

Notice that the current type of y was B initially, then the TSMI-based optimisation changed it to the more specific type C, and then the dynamic loading of class D changed the current type of y back to B.

In summary, the new ideas are:

- A type is a pair of Java types in which the second Java type is a subtype of the first Java type.
- The Null type is used to type dynnew.
- A type cast uses the first Java type in the pair.

Our main theorem is that with a type system based on those three ideas, TSMI-based devirtualisation and revirtualisation is type preserving. As our experiments in the previous section show, the new approach will lead to considerably more inlining than the previously best approach, namely CHA. Later we formalise our ideas and prove the main theorem. First we clarify how revirtualisation is done and how we formalise it, and clarify how we do our proofs.

Patch Construct. Until now we have not said much about how a virtual machine revirtualises a method invocation. The main problem with revirtualisation is that an invalidated method inlining may be in a currently executing method, requiring a nontrivial update of the program state. We focus on a technique for doing this update called *patching*, used by some virtual machines (for example [14] and ORP [5,6]). Patching is a form of in-place code modification for reverting to unoptimised code, and does not require any update of the stack or recompilation of methods. The basic idea is to compile the call `x.m()` to the following code:

```
label l1:  [Inline x.C::m()]
label l3:  ...
label l2:  x.m();      [out of line]
           jump l3;
```

(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 l2;` instruction at address `l1`. There are important low-level details that we abstract (these and techniques other than patching are described in our previous paper [10]).

To formalise this idea in a small language, we need an expression of the form $\mathtt{e}_1\ \mathtt{patchto}^{\ell}\ \mathtt{e}_2$ where ℓ is a label. Additionally, program states will have a component, called the patch set, that is a set of labels of patches that have been applied. If ℓ is in this set then the above expression acts like $\mathtt{e}_2$, if not it acts like $\mathtt{e}_1$. This idea models what the assembly sequence above does.

Note that, as in previous papers, we concentrate on devirtualisation, the first step of method inlining, as the other step is straightforward. Given this focus, a general patch construct is not needed. Instead we use a construct of the form $\mathtt{e.[C::]}^{\ell}\mathtt{m()}$, which can be though of as $\mathtt{e.C::m()}\ \mathtt{patchto}^{\ell}\ \mathtt{e.m()}$ where `e.C::m()` invokes `C`'s implementation of `m` on `e`, and ultimately should be thought of as the code above.

The correctness of speculative inlining with patching is far less obvious than the correctness of inlining for whole programs. We use a proof framework developed in our previous paper [10]. Note that we do devirtualisation of both the initial program and of dynamically loaded classes. Furthermore, the patching operation, which is part of the optimisation, is a runtime operation. The usual formalisation methods do not suffice, and instead we formalise the optimisation as a second semantics. This semantics includes the transformation that does devirtualisation and the patching operation as part of the semantics of `dynnew`. To prove correctness of the optimisation we show that the optimising semantics gives the same meaning to a program as a standard semantics does. To prove type preservation, we prove the optimising semantics type safe.

4 Dynamic Loading Language

This section begins the formal development of our results. It defines a simple language with dynamic class loading that is the source language for the optimisation. The language is a variant of Featherweight Java (FJ [13]), adding just one new expression form for dynamically loading a new class. Due to space limitations we omit many standard or obvious details (readers can refer to the original FJ paper or our previous dynamic loading paper). The optimised code will use a slightly different syntax (see the following section), here is the common syntax:

Expressions	$\mathtt{e}$	$::= \mathtt{x}^\ell \mid \mathtt{new}\ \mathtt{C}^\ell\mathtt{(\overline{e})} \mid \mathtt{e.f}^\ell \mid \mathtt{e.m}^\ell\mathtt{(\overline{e})} \mid \mathtt{(t)}^\ell\mathtt{e} \mid \mathtt{dynnew}^\ell$
Method Declarations	$\mathtt{M}$	$::= \mathtt{t}^\ell\ \mathtt{m(\overline{t}\ \overline{x^\ell})}\ \{\ \mathtt{return\ e;}\ \}$
Class Declarations	$\mathtt{CD}$	$::= \mathtt{class}\ \mathtt{C}_1\ \mathtt{extends}\ \mathtt{C}_2\ \{\ \overline{\mathtt{t}}\ \overline{\mathtt{f}^\ell}\mathtt{;}\ \overline{\mathtt{M}}\ \}$

And here is the standard syntax:

Types	$\mathtt{t} ::= \mathtt{C}$
Program State	$\mathtt{P} ::= \mathtt{(\overline{CD};e)}$

We use standard metavariables and the bar notation from the FJ paper.

To simplify matters, we assume that field names are unique, that all $\mathtt{x}^\ell$ expressions have the same label as the binder of $\mathtt{x}$, and that all labels of `this` in a class have the same label. These restrictions mean that $\mathit{lab}(\mathtt{f})$ identifies a unique label for each field declared in a program, and that in the given scope $\mathit{lab}(\mathtt{x})$ identifies a unique label for each variable in that scope.

Some auxiliary definitions that are used in the rest of the paper appear in appendix A. The standard operation semantics is similar to FJ extended with a rule for `dynnew`:

$$\frac{\mathtt{CD} = \mathtt{class\ C\ extends}\ \cdots\ \{\ \cdots\ \}}{(\overline{\mathtt{CD}};\mathtt{X}\langle\mathtt{dynnew}^\ell\rangle) \overset{\mathtt{CD},\overline{\mathtt{e}},\ell'}{\longmapsto_{\mathtt{s}}} (\overline{\mathtt{CD}},\mathtt{CD};\mathtt{X}\langle\mathtt{new\ C}^{\ell'}\mathtt{(\overline{e})}\rangle)} \quad (1)$$

Here $\mathtt{X}$ ranges over evaluation contexts. To keep the semantics deterministic, we explicitly label the reduction with a label of the form $(\mathtt{CD},\overline{\mathtt{e}},\ell)$, where $\mathtt{CD}$ is the newly loaded class, $\overline{\mathtt{e}}$ are the initialiser expressions, and ℓ is the label to use on the new object.

The typing rules are those of Featherweight Java extended with a rule for `dynnew`; they can be recovered from the more general rules in Figure 4 by ignoring the right type in the type pairs. The type system is sound as can be proven by standard techniques.

$$\mathit{poly}(\mathtt{P},\phi) = \{\ell \mid \mathtt{e.[C::]}^{\ell}\mathtt{m}(\overline{\mathtt{e}}) \in \mathtt{P}, \exists \mathtt{D} \in \phi(\mathit{lab}(\mathtt{e})) : \mathit{impl}(\mathtt{P},\mathtt{D},\mathtt{m}) \neq \mathtt{C::m}\}$$

$$\frac{\mathit{fields}(\overline{\mathtt{CD}},\mathtt{C}) = \overline{\mathtt{t}}\ \overline{\mathtt{f}};}{(\overline{\mathtt{CD}};\mathtt{S};\mathtt{X}\langle\mathtt{new}\ \mathtt{C}^{\ell_1}(\overline{\mathtt{e}}).\mathtt{f}_i^{\ell_2}\rangle) \mapsto_o (\overline{\mathtt{CD}};\mathtt{S};\mathtt{X}\langle\mathtt{e}_i\rangle)} \quad (2)$$

$$\frac{\mathit{mbody}(\overline{\mathtt{CD}},\mathtt{C},\mathtt{m}) = (\overline{\mathtt{x}},\mathtt{e},\ell)}{(\overline{\mathtt{CD}};\mathtt{S};\mathtt{X}\langle\mathtt{new}\ \mathtt{C}^{\ell_1}(\overline{\mathtt{e}}).\mathtt{m}^{\ell_2}(\overline{\mathtt{d}})\rangle) \mapsto_o (\overline{\mathtt{CD}};\mathtt{S};\mathtt{X}\langle\mathtt{e}\{\mathtt{this},\overline{\mathtt{x}} := \mathtt{new}\ \mathtt{C}^{\ell_1}(\overline{\mathtt{e}}),\overline{\mathtt{d}}\}\rangle)} \quad (3)$$

$$\frac{\overline{\mathtt{CD}} \vdash \mathtt{C} <: \mathtt{D}}{(\overline{\mathtt{CD}};\mathtt{S};\mathtt{X}\langle(\mathtt{(D,E)})^{\ell'}\mathtt{new}\ \mathtt{C}^{\ell}(\overline{\mathtt{e}})\rangle) \mapsto_o (\overline{\mathtt{CD}};\mathtt{S};\mathtt{X}\langle\mathtt{new}\ \mathtt{C}^{\ell}(\overline{\mathtt{e}})\rangle)} \quad (4)$$

$$\frac{\begin{array}{c}\mathtt{CD} = \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \cdots\ \{\ \cdots\ \} \qquad \mathtt{P} = (\overline{\mathtt{CD}},\mathtt{CD};\mathtt{S};\mathtt{X}\langle\mathtt{new}\ \mathtt{C}^{\ell}(\overline{\mathtt{e}})\rangle) \qquad \phi = \mathit{fa}(\mathtt{P}) \\ \overline{\mathtt{CD}'} = \mathit{retype}(\overline{\mathtt{CD}},\phi) \qquad \mathtt{X}' = \mathit{retype}(\mathtt{X},\phi) \qquad \mathtt{CD}' = [\![\mathit{retype}(\mathtt{CD},\phi)]\!]_{\overline{\mathtt{CD}},\mathtt{CD},\phi} \\ \overline{\mathtt{e}'} = [\![\mathit{retype}(\overline{\mathtt{e}},\phi)]\!]_{\overline{\mathtt{CD}},\mathtt{CD},\phi} \qquad \mathtt{S}' = \mathtt{S} \cup \mathit{poly}(\mathtt{P},\phi)\end{array}}{(\overline{\mathtt{CD}};\mathtt{S};\mathtt{X}\langle\mathtt{dynnew}^{\ell}\rangle) \overset{\mathtt{CD},\overline{\mathtt{e}},\ell'}{\mapsto_o} (\overline{\mathtt{CD}'},\mathtt{CD}';\mathtt{S}';\mathtt{X}'\langle\mathtt{new}\ \mathtt{C}^{\ell'}(\overline{\mathtt{e}'})\rangle)} \quad (5)$$

$$\frac{\mathit{mbody}(\overline{\mathtt{CD}},\left\{\begin{array}{ll}\mathtt{C} & \ell_2 \in \mathtt{S} \\ \mathtt{D} & \ell_2 \notin \mathtt{S}\end{array}\right\},\mathtt{m}) = (\overline{\mathtt{x}},\mathtt{e},\ell)}{(\overline{\mathtt{CD}};\mathtt{S};\mathtt{X}\langle\mathtt{new}\ \mathtt{C}^{\ell_1}(\overline{\mathtt{e}}).\mathtt{[D::]}^{\ell_2}\mathtt{m}(\overline{\mathtt{d}})\rangle) \mapsto_o (\overline{\mathtt{CD}};\mathtt{S};\mathtt{X}\langle\mathtt{e}\{\mathtt{this},\overline{\mathtt{x}} := \mathtt{new}\ \mathtt{C}^{\ell_1}(\overline{\mathtt{e}}),\overline{\mathtt{d}}\}\rangle)} \quad (6)$$

Fig. 3. Optimised Operational Semantics

5 Devirtualisation Optimisation

This section formalises speculative devirtualisation with patching for revirtualisation as a second semantics, called the *optimising semantics*, for the language of the previous section. The additional constructs required are described next, following by the actual transformation, and finally the semantics and the type system.

Syntax. The optimised semantics needs a patching construct and an associated patch set in the program states, and two types in each static typing annotation—the original and the current type. The modified syntax is:

Types	$\mathtt{t} ::= (\mathtt{C}_1,\mathtt{C}_2)$
Expressions	$\mathtt{e} ::= \cdots \mid \mathtt{e.[C::]}^{\ell}\mathtt{m}(\overline{\mathtt{e}})$
Program States	$\mathtt{P} ::= (\overline{\mathtt{CD}};\mathtt{S};\mathtt{e})$

Here S, called the *patch set*, is the set of labels of the patch constructs that had to be revirtualised. A patch construct has the form $\mathtt{e.[C::]}^{\ell}\mathtt{m}(\overline{\mathtt{e}})$. If ℓ is in the patch set S then this expression acts like a normal virtual method invocation $\mathtt{e.m}^{\ell}(\overline{\mathtt{e}})$. Otherwise it acts like a nonvirtual method invocation—it invokes C's version of m on object e with arguments $\overline{\mathtt{e}}$. Types are now pairs where the left class name is the original type from the unoptimised code, and the right class name is the current type based on the current flow analysis.

Transformation. The transformation of code is based on a *flow* that assigns sets of class names, called *flow sets*, to expressions, fields, method parameters, and method returns. The set should include all classes in the current program state that the expression might evaluate to. A flow analysis takes a program state and returns a flow for it, and it should ignore the current types. Before applying the transformation, the static type information must be transformed so that the current types reflect the flow used. The *retype* function achieves this change. Its definition is in Appendix A, as the only interesting clause is: $\mathit{retype}((\mathtt{C}_1,\mathtt{C}_2)^{\ell}, \phi) = (\mathtt{C}_1, \sqcup\, \phi(\ell))$. The transformation takes an expression, method declaration, or class declaration and changes monomorphic virtual method invocations into patchable nonvirtual method invocations. It appears in Appendix A as the only interesting clause is:

$$[\![\mathtt{e.m}^{\ell}(\overline{\mathtt{e}})]\!]_{\overline{\mathtt{CD}},\phi} = [\![\mathtt{e}]\!]_{\overline{\mathtt{CD}},\phi}.[\mathtt{C::}]^{\ell}\mathtt{m}([\![\overline{\mathtt{e}}]\!]_{\overline{\mathtt{CD}},\phi}) \quad \text{if } \forall \mathtt{D} \in \phi(\mathit{lab}(\mathtt{e})) : \mathit{impl}(\overline{\mathtt{CD}}, \mathtt{D}, \mathtt{m}) = \mathtt{C::m}$$

Optimised Semantics. The optimised semantics is parameterised by a flow analysis *fa* (that is, a function that takes an optimised-syntax program state and returns a flow for it). A standard syntax program $(\overline{\mathtt{CD}};\mathtt{e})$ starts in the optimised semantics state $([\![\mathit{retype}(\overline{\mathtt{CD}}, \phi)]\!]_{\overline{\mathtt{CD}},\phi};\emptyset;[\![\mathit{retype}(\mathtt{e}, \phi)]\!]_{\overline{\mathtt{CD}},\phi})$ where $\phi = \mathit{fa}(\overline{\mathtt{CD}};\emptyset;\mathtt{e})$. In other words a flow analysis is performed on the initial program and used to transform it to form the initial state along with an empty patch set.

The reduction rules for the optimised semantics appear in Figure 3. The rules are similar to the standard semantics with the following modifications. The rule for cast uses the original type in the cast rather than the current type to determine if the cast should succeed. The rule for dynamic new is the most complex. It performs a flow analysis on the unoptimised new program state. Then it uses this flow analysis to retype the program state and to transform the new class declaration and initialiser expressions. Finally, it adds to the patch set the labels of patch constructs that are no longer monomorphic. The rule for the patch construct is similar to the rule for method invocation except in how it finds the method body. If the label is in the patch set, then the construct is "patched" and should act like a virtual method invocation. In this case it uses the object's class to lookup the body as in the rule for method invocation. If the label is not in the patch set, then the construct acts like a nonvirtual invocation, and uses the class in the construct, `D`, to lookup the method body.

Type System. The typing rules appear in Figure 4. The rules are fairly straightforward. They essentially are checking the original and current typing in parallel. To look up field or method types, since these are the same whether we look in the superclass or subclass, we simply use the original type. Two rules treat the current and original types differently. For dynamic new, the current is `Null` as it is always retyped before it is replaced by an actual object, but its original type must be `Object`. For the patching construct, if not currently patched then the object must be in the type `E` being dispatched to, so we require the current type to be a subtype of this.

Except for the details of subtyping, the rules are deterministic, and for a program state `P`, there is a unique `t` and derivation of $\vdash \mathtt{P} \in \mathtt{t}$. Therefore, given a program and an occurrence of a label in it, there is a uniquely determined type associated with that occurrence: either the type of the expression it labels, or

$$\frac{}{\overline{\texttt{CD}} \vdash \texttt{Null <: Object} \qquad \overline{\texttt{CD}} \vdash \texttt{Object <: Object}} \tag{7}$$

$$\frac{\texttt{class C extends D} \; \{ \; \cdots \; \} \in \overline{\texttt{CD}}}{\overline{\texttt{CD}} \vdash \texttt{Null <: C} \qquad \overline{\texttt{CD}} \vdash \texttt{C <: C} \qquad \overline{\texttt{CD}} \vdash \texttt{C <: D}} \tag{8}$$

$$\frac{\overline{\texttt{CD}} \vdash \texttt{C <: D} \qquad \overline{\texttt{CD}} \vdash \texttt{D <: E}}{\overline{\texttt{CD}} \vdash \texttt{C <: E}} \tag{9}$$

$$\frac{\overline{\texttt{CD}} \vdash \texttt{C}_2 \texttt{ <: } \texttt{C}_1}{\overline{\texttt{CD}} \vdash \texttt{(C}_1\texttt{,C}_2\texttt{)}} \tag{10}$$

$$\frac{\overline{\texttt{CD}} \vdash \texttt{C}_1 \texttt{ <: } \texttt{D}_1 \qquad \overline{\texttt{CD}} \vdash \texttt{C}_2 \texttt{ <: } \texttt{D}_2}{\overline{\texttt{CD}} \vdash \texttt{(C}_1\texttt{,C}_2\texttt{) <: (D}_1\texttt{,D}_2\texttt{)}} \tag{11}$$

$$\frac{}{\overline{\texttt{CD}}; \texttt{S}; \Gamma \vdash \texttt{x} \in \Gamma(\texttt{x})} \tag{12}$$

$$\frac{\mathit{fields}(\overline{\texttt{CD}}, \texttt{C}) = \overline{\texttt{t}}\;\overline{\texttt{f}}\texttt{;} \qquad \overline{\texttt{CD}}; \texttt{S}; \Gamma \vdash \overline{\texttt{e}} \in \overline{\texttt{t}'} \qquad \overline{\texttt{CD}} \vdash \overline{\texttt{t}'} \texttt{ <: } \overline{\texttt{t}}}{\overline{\texttt{CD}}; \texttt{S}; \Gamma \vdash \texttt{new C}^{\ell}\texttt{(}\overline{\texttt{e}}\texttt{)} \in \texttt{(C,C)}} \tag{13}$$

$$\frac{\overline{\texttt{CD}}; \texttt{S}; \Gamma \vdash \texttt{e} \in \texttt{(C,D)} \qquad \mathit{fields}(\overline{\texttt{CD}}, \texttt{C}) = \overline{\texttt{t}}\;\overline{\texttt{f}}\texttt{;}}{\overline{\texttt{CD}}; \texttt{S}; \Gamma \vdash \texttt{e.f}_i^{\ell} \in \texttt{t}_i} \tag{14}$$

$$\frac{\overline{\texttt{CD}}; \texttt{S}; \Gamma \vdash \texttt{e} \in \texttt{(C,D)} \qquad \mathit{mtype}(\overline{\texttt{CD}}, \texttt{C}, \texttt{m}) = \overline{\texttt{t}} \to \texttt{t} \qquad \overline{\texttt{CD}}; \texttt{S}; \Gamma \vdash \overline{\texttt{e}} \in \overline{\texttt{t}'} \qquad \overline{\texttt{CD}} \vdash \overline{\texttt{t}'} \texttt{ <: } \overline{\texttt{t}}}{\overline{\texttt{CD}}; \texttt{S}; \Gamma \vdash \texttt{e.m}^{\ell}\texttt{(}\overline{\texttt{e}}\texttt{)} \in \texttt{t}} \tag{15}$$

$$\frac{\overline{\texttt{CD}}; \texttt{S}; \Gamma \vdash \texttt{e} \in \texttt{t}' \qquad \overline{\texttt{CD}} \vdash \texttt{t}}{\overline{\texttt{CD}}; \texttt{S}; \Gamma \vdash \texttt{(t)}^{\ell}\texttt{e} \in \texttt{t}} \tag{16}$$

$$\frac{}{\overline{\texttt{CD}}; \texttt{S}; \Gamma \vdash \texttt{dynnew}^{\ell} \in \texttt{(Object,Null)}} \tag{17}$$

$$\frac{\begin{array}{c}\overline{\texttt{CD}}; \texttt{S}; \Gamma \vdash \texttt{e} \in \texttt{(C,D)} \\ \mathit{mtype}(\overline{\texttt{CD}}, \texttt{C}, \texttt{m}) = \overline{\texttt{t}} \to \texttt{t} \\ \overline{\texttt{CD}}; \texttt{S}; \Gamma \vdash \overline{\texttt{e}} \in \overline{\texttt{t}'} \\ \overline{\texttt{CD}} \vdash \overline{\texttt{t}'} \texttt{ <: } \overline{\texttt{t}} \\ \mathit{mtype}(\overline{\texttt{CD}}, \texttt{E}, \texttt{m}) \text{ is defined} \\ \ell \notin \texttt{S} \Rightarrow \overline{\texttt{CD}} \vdash \texttt{D <: E}\end{array}}{\overline{\texttt{CD}}; \texttt{S}; \Gamma \vdash \texttt{e.[E::]}^{\ell}\texttt{m(}\overline{\texttt{e}}\texttt{)} \in \texttt{t}} \tag{18}$$

$$\frac{\begin{array}{c}\overline{\texttt{CD}} \vdash \texttt{t} \qquad \overline{\texttt{CD}} \vdash \overline{\texttt{t}} \\ \overline{\texttt{CD}}; \texttt{S}; \texttt{this : (C,C)}, \overline{\texttt{x}} : \overline{\texttt{t}} \vdash \texttt{e} \in \texttt{t}' \qquad \overline{\texttt{CD}} \vdash \texttt{t}' \texttt{ <: t} \\ \mathit{can\text{-}declare}(\overline{\texttt{CD}}, \texttt{C}, \texttt{m}, \overline{\texttt{t}} \to \texttt{t})\end{array}}{\overline{\texttt{CD}}; \texttt{S} \vdash \texttt{t}^{\ell} \; \texttt{m(}\overline{\texttt{t}}\;\overline{\texttt{x}^{\ell}}\texttt{)} \; \{ \; \texttt{return e;} \; \} \text{ in } \texttt{C}} \tag{19}$$

$$\frac{\overline{\texttt{CD}} \vdash \overline{\texttt{t}} \qquad \overline{\texttt{CD}}; \texttt{S} \vdash \overline{\texttt{M}} \text{ in } \texttt{C}}{\overline{\texttt{CD}}; \texttt{S} \vdash \texttt{class C extends D} \; \{ \; \overline{\texttt{t}}\;\overline{\texttt{f}^{\ell}}\texttt{;} \; \overline{\texttt{M}} \; \}} \tag{20}$$

$$\frac{\overline{\texttt{CD}}; \texttt{S} \vdash \overline{\texttt{CD}} \qquad \overline{\texttt{CD}}; \texttt{S}; \cdot \vdash \texttt{e} \in \texttt{t}}{\vdash \texttt{(}\overline{\texttt{CD}}\texttt{;S;e)} \in \texttt{t}} \tag{21}$$

Fig. 4. Typing Rules for the Optimised Syntax

the field, return, or parameter type that it labels. A flow ϕ for a program is *type respecting* if and only if for each label ℓ in the program, each class C in $\phi(\ell)$, and each original type D associated with ℓ, C is a subtype of D.

6 Correctness

In this section we prove the optimisation correct, that is, that it preserves typability and operational semantics. The optimisation is correct, however, only for certain flow analyses—the ones that respect the typing rules and approximate the operational semantics. A flow ϕ for a program $\mathtt{P}$ is *acceptable* exactly when it satisfies the conditions in Figure 5. A flow analysis *fa* is correct if $fa(\mathtt{P})$ is an acceptable and type-respecting flow for $\mathtt{P}$ whenever $\vdash \mathtt{P} \in \mathtt{t}$ for some $\mathtt{t}$. We prove the optimisation correct when it is based on a correct flow analysis.

Typability Preservation. Since the optimisation is stated as a second semantics for the language, typability preservation means that a well-typed standard syntax program does not get stuck in the optimised semantics. However, it is not enough that the original program type checks, all dynamically loaded classes must type check as well. We say that $(\mathtt{CD}, \overline{\mathtt{e}}, \ell)$ type checks with respect to program $(\overline{\mathtt{CD}};\mathtt{S};\mathtt{e})$ exactly when $\overline{\mathtt{CD}}, \mathtt{CD}; \mathtt{S} \vdash \mathtt{CD}$ and $\overline{\mathtt{CD}}, \mathtt{CD}; \mathtt{S}; \cdot \vdash \overline{\mathtt{e}} \in \overline{\mathtt{t}}$ where $\mathtt{CD} = \mathtt{class\ C\ extends} \cdots \{ \cdots \}$ and $fields(\overline{\mathtt{CD}}, \mathtt{CD}, \mathtt{C}) = \overline{\mathtt{t}}\ \overline{\mathtt{f}}\mathtt{;}$. We say that a reduction sequence type checks exactly when the initial program state type checks and all the labels in the reduction sequence type check with respect to the program state that precedes them.

Theorem 1 (Typability Preservation). *If* $\mathtt{P}$ *is a well-typed standard-syntax program, then any well-typed reduction sequence in the optimised semantics, which starts from a state corresponding to* $\mathtt{P}$ *and is based on a correct flow analysis, does not end in a stuck state.*

The proof is given in the full version of the paper, which is available from the webpage `http://www.cs.ucla.edu/~palsberg/publications.html`. The key to proving the theorem is proving that at each point in the reduction sequence the program state type checks and there is an acceptable and type-respecting flow for the program state. Formally, we define $\vdash (\mathtt{P}, \phi)\ \mathsf{good}$ to mean $\vdash \mathtt{P} \in \mathtt{t}$ for some $\mathtt{t}$, ϕ is an acceptable and type-respecting flow for $\mathtt{P}$, and the current type of every static typing annotation in $\mathtt{P}$ is $\sqcup\phi(\ell)$ where ℓ is the label associated with the annotation. As with standard type soundness arguments, we show that reduction preserves goodness (rather than typability), and that typable (a subset of good) states are not stuck.

Operational Correctness. We prove that the optimisation preserves semantics, specifically that the optimised semantics simulates the standard semantics and vice versa. To state the result we need a *correspondence* relation $corresponds_\phi$ $(\mathtt{P}, \mathtt{P}')$. 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 the full version of the paper. Essentially, where the left program has a virtual dispatch the right program may have one of two expressions. It can have a corresponding virtual dispatch. It can also have an equivalent patch construct if the virtual dispatch is monomorphic in the current

– For each $\texttt{new C}^\ell\texttt{(}\overline{\texttt{e}}\texttt{)}$ in $\texttt{P}$ where $\mathit{fields}(\overline{\texttt{CD}}, \texttt{C}) = \overline{\texttt{t}}\,\overline{\texttt{f}}\texttt{;}$:

$$\phi(\mathit{lab}(\overline{\texttt{e}})) \subseteq \phi(\mathit{lab}(\overline{\texttt{f}})) \tag{22}$$
$$\texttt{C} \in \phi(\ell) \tag{23}$$

– For each $\texttt{e.f}^\ell$ in $\texttt{P}$:

$$\phi(\mathit{lab}(\texttt{f})) = \phi(\ell) \tag{24}$$

– For each $\texttt{e.m}^\ell\texttt{(}\overline{\texttt{e}}\texttt{)}$ in $\texttt{P}$ where $\texttt{e}$ has type $\texttt{(C}_1\texttt{,C}_2\texttt{)}$ and $\mathit{mbody}(\texttt{P}, \texttt{C}_1, \texttt{m}) = (\overline{\texttt{x}}, \texttt{e}', \ell')$:

$$\phi(\mathit{lab}(\overline{\texttt{e}})) \subseteq \phi(\mathit{lab}(\overline{\texttt{x}})) \tag{25}$$
$$\phi(\ell') = \phi(\ell) \tag{26}$$

And for each $\texttt{D} \in \phi(\mathit{lab}(\texttt{e}))$, $\mathit{impl}(\texttt{P}, \texttt{D}, \texttt{m}) = \texttt{E::m}$, and ℓ' labels $\texttt{this}$ in $\texttt{E}$:

$$\phi(\mathit{lab}(\texttt{e})) \subseteq \phi(\ell') \tag{27}$$

– For each $\texttt{((C,D))}^\ell\texttt{e}$ in $\texttt{P}$:

$$\phi(\mathit{lab}(\texttt{e})) \cap \mathit{subclasses}(\texttt{P}, \texttt{C}) \subseteq \phi(\ell) \tag{28}$$

– For each $\texttt{dynnew}^\ell$ in $\texttt{P}$:

$$\phi(\ell) = \emptyset \tag{29}$$

– For each $\texttt{e.[C::]}^\ell\texttt{m(}\overline{\texttt{e}}\texttt{)}$ in $\texttt{P}$ where $\texttt{e}$ has type $\texttt{(C}_1\texttt{,C}_2\texttt{)}$ and $\mathit{mbody}(\texttt{P}, \texttt{C}_1, \texttt{m}) = (\overline{\texttt{x}}, \texttt{e}', \ell')$:

$$\phi(\mathit{lab}(\overline{\texttt{e}})) \subseteq \phi(\mathit{lab}(\overline{\texttt{x}})) \tag{30}$$
$$\phi(\ell') = \phi(\ell) \tag{31}$$

And if $\ell \in \texttt{S}$ where $\texttt{P} = \texttt{(}\cdots\texttt{;S;}\cdots\texttt{)}$ then for each $\texttt{D} \in \phi(\mathit{lab}(\texttt{e}))$, $\mathit{impl}(\texttt{P}, \texttt{D}, \texttt{m}) = \texttt{E::m}$, and ℓ' labels $\texttt{this}$ in $\texttt{E}$:

$$\phi(\mathit{lab}(\texttt{e})) \subseteq \phi(\ell') \tag{32}$$

And if $\ell \notin \texttt{S}$ then the following where $\mathit{impl}(\texttt{P}, \texttt{C}, \texttt{m}) = \texttt{E::m}$ and ℓ' labels $\texttt{this}$ in $\texttt{E}$:

$$\phi(\mathit{lab}(\texttt{e})) \subseteq \phi(\ell') \tag{33}$$

– For each class $\texttt{C}$ in $\texttt{P}$ with label ℓ for $\texttt{C}$'s $\texttt{this}$ occurrences:

$$\texttt{C} \in \phi(\ell) \tag{34}$$

– For each method $\texttt{t}^\ell\ \texttt{m(}\overline{\texttt{t}}\ \overline{\texttt{x}^{\ell}}\texttt{)}\ \{\ \texttt{return e;}\ \}$ in $\texttt{P}$:

$$\phi(\mathit{lab}(\texttt{e})) \subseteq \phi(\ell) \tag{35}$$

– If $\texttt{t}^{\ell_1}\ \texttt{m(}\bar{t}\ \overline{x_1^{\ell_1}}\texttt{)}\ \{\ \texttt{return e}_1\texttt{;}\ \}$ overrides $\texttt{t}^{\ell_2}\ \texttt{m(}\bar{t}\ \overline{x_2^{\ell_2}}\texttt{)}\ \{\ \texttt{return e}_2\texttt{;}\ \}$ in $\texttt{P}$ then:

$$\phi(\ell_1) = \phi(\ell_2) \tag{36}$$
$$\phi(\overline{\ell_1}) = \phi(\overline{\ell_2}) \tag{37}$$

Fig. 5. The Conditions for an Acceptable Flow Analysis

program (the subscripts $\overline{\mathtt{CD}}$ and ϕ on the relation) or if the patch label is in the current patch set (the subscript $\mathtt{S}$ on the relation).

Given the correspondence relation, two facts are true. First, if $\mathtt{P}'$ is the initial state in the optimised semantics for program $\mathtt{P}$ then $\mathit{corresponds}_{\phi}(\mathtt{P}, \mathtt{P}')$ where ϕ is the flow analysis used to compute the initial state. Second, the optimised semantics simulates the standard semantics and vice versa, as stated in the following theorem.

Theorem 2 (Operational Correctness).
If $\mathit{corresponds}_{\phi_1}(\mathtt{P}_1, \mathtt{P}'_1)$ *and the flow-analysis is correct then:*

- *If* $\mathtt{P}_1 \overset{L}{\longmapsto}_{\mathtt{s}} \mathtt{P}_2$ *then* $\mathtt{P}'_1 \overset{L}{\longmapsto}_{\mathtt{o}} \mathtt{P}'_2$ *and* $\mathit{corresponds}_{\phi_2}(\mathtt{P}_2, \mathtt{P}'_2)$ *for some* $\mathtt{P}'_2$ *and* ϕ_2*.*
- *If* $\mathtt{P}'_1 \overset{L}{\longmapsto}_{\mathtt{o}} \mathtt{P}'_2$ *then* $\mathtt{P}_1 \overset{L}{\longmapsto}_{\mathtt{s}} \mathtt{P}_2$ *and* $\mathit{corresponds}_{\phi_2}(\mathtt{P}_2, \mathtt{P}'_2)$ *for some* $\mathtt{P}_2$ *and* ϕ_2*.*

The proof of both these facts is very similar to the proof in our previous paper [10].

7 Conclusion

We presented a new type system and theorem that shows that TSMI is type preserving in the presence of dynamic class loading. Our experimental results show that TSMI leads to considerably more inlining than the current best approach, namely CHA. Our experimental results also show the value of analyzing all local plugins at start-up time: only few inlinings will be invalidated when loading a local plugin. The flow analysis has to be recomputed only when a plugin is loaded from non-local sources. Since such remote operations involve considerable delay anyway, the extra delay from redoing the flow analysis is unlikely to be noticable.

Researchers have recently developed many new ideas for efficiently doing flow analysis, virtualisation, and devirtualisation in JIT compilers [4,12,19,20]. Our results can form the basis of a new generation of typed intermediate representations used by powerful, type-preserving JIT compilers.

In future work we would like to go beyond the static counts of virtual call sites. We would like to count how many times each call site is executed, and count how many call sites turn out to be monomorphic at run time. Researchers might also explore how our results fit with recent work on dynamic code updates [23].

References

1. B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno virtual machine. *IBM System Journal*, 39(1), February 2000.

2. David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of OOPSLA'96, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 324–341, 1996.
3. David Francis Bacon. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages.* PhD thesis, Computer Science Division, University of California, Berkeley, December 1997. Report No. UCB/CSD-98-1017.
4. Jeff Bogda and Ambuj K. Singh. Can a shape analysis work at run-time? In *Java Virtual Machine Research and Technology Symposium*, 2001.
5. 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.
6. 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, June 2000.
7. J. Dean and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. Technical Report 94-12-01, Department of Computer Science, University of Washington at Seattle, December 1994.
8. J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In W. Olthoff, editor, *Proceedings of the Ninth European Conference on Object-Oriented Programming (ECOOP'95)*, pages 77–101, Aarhus, Denmark, August 1995. Springer-Verlag.
9. 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, 2000.
10. Neal Glew and Jens Palsberg. Method inlining, dynamic class loading, and type soundness. *Journal of Object Technology.* Preliminary version in Sixth Workshop on Formal Techniques for Java-like Programs, Oslo, Norway, June 2004.
11. Neal Glew and Jens Palsberg. Type-safe method inlining. *Science of Computer Programming*, 52:281–306, 2004. Preliminary version in Proceedings of ECOOP'02, European Conference on Object-Oriented Programming, pages 525–544, Springer-Verlag (*LNCS* 2374), Malaga, Spain, June 2002.
12. Martin Hirzel, Amer Diwan, and Michael Hind. Pointer analysis in the presence of dynamic class loading. In *Proceedings of ECOOP'04, 16th European Conference on Object-Oriented Programming*, pages 96–122, 2004.
13. Atsushi Igarashi, Benjamion 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.
14. 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 OOPSLA'00, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 294-310, 2000.
15. Christopher League, Zhong Shao, and Valery Trifonov. Representing Java classes in a typed intermediate language. In *Proceedings of ICFP '99, ACM SIGPLAN International Conference on Functional Programming*, pages 183–196, 1999.
16. 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*, 1996.

17. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.
18. Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *Proceedings of OOPSLA'91, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 146–161, 1991.
19. Igor Pechtchanski and Vivek Sarkar. Dynamic optimistic interprocedural analysis: A framework and an application. In *Proceedings of OOPSLA'01, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 195–210, 2001.
20. Feng Qian and Laurie J. Hendren. Towards dynamic interprocedural analysis in JVMs. In *Virtual Machine Research and Technology Symposium*, pages 139–150, 2004.
21. Olin Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, CMU, May 1991. CMU–CS–91–145.
22. 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.
23. Gareth Stoyle, Michael W. Hicks, Gavin M. Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis mutandis: Safe and predictable dynamic software updating. In *Proceedings of POPL'05, SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 183–194, 2005.
24. 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.
25. P.Y. Wei. A brief overview of the VIOLA engine, and its applications. `http://www.xcf.berkeley.edu/~wei/viola/violaIntro.html`, 1994.
26. 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.

Appendix A: Details of the Formalisation

The function $\mathit{fields}(\overline{\mathtt{CD}}, \mathtt{C})$ returns $\mathtt{C}$'s fields (declared and inherited) and their types; $\mathit{mtype}(\overline{\mathtt{CD}}, \mathtt{C}, \mathtt{m})$ returns the signature of $\mathtt{m}$ in $\mathtt{C}$, it has the form $\overline{\mathtt{t}} \to \mathtt{t}$ where $\overline{\mathtt{t}}$ are the argument types and $\mathtt{t}$ is the return type; $\mathit{mbody}(\overline{\mathtt{CD}}, \mathtt{C}, \mathtt{m})$ returns the implementation of $\mathtt{m}$ in $\mathtt{C}$, it has the form $(\overline{\mathtt{x}}, \mathtt{e}, \ell)$ where $\mathtt{e}$ is the expression to evaluate, $\overline{\mathtt{x}}$ are the parameters, and ℓ is the label of the method return; $\mathit{impl}(\overline{\mathtt{CD}}, \mathtt{C}, \mathtt{m})$ returns the class from which $\mathtt{C}$ inherits $\mathtt{m}$ (this could be $\mathtt{C}$ itself), it has the form $\mathtt{D}::\mathtt{m}$ where $\mathtt{D}$ is the class; $\mathit{can\text{-}declare}(\overline{\mathtt{CD}}, \mathtt{C}, \mathtt{m}, \overline{\mathtt{t}} \to \mathtt{t})$ checks that $\mathtt{C}$ is allowed to declare $\mathtt{m}$ with signature $\overline{\mathtt{t}} \to \mathtt{t}$—this would not be the case if one of $\mathtt{C}$'s ancestors in the class hierarchy also declared $\mathtt{m}$ with a different signature.

Field Lookup, Method Information and Inheritance Checking

$$\frac{}{\mathit{fields}(\overline{\mathtt{CD}}, \mathtt{Object}) = \cdot} \qquad \frac{\overline{\mathtt{CD}}(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\ \overline{\mathtt{t}\ \mathtt{f}}^{\ell}\,;\ \overline{\mathtt{M}}\ \} \quad \mathit{fields}(\overline{\mathtt{CD}}, \mathtt{D}) = \overline{\mathtt{t}'\ \mathtt{f}'};}{\mathit{fields}(\overline{\mathtt{CD}}, \mathtt{C}) = \overline{\mathtt{t}'\ \mathtt{f}'};\overline{\mathtt{t}\ \mathtt{f}}\,;}$$

$$\frac{\overline{\mathtt{CD}}(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\ \overline{\mathtt{t}\ \mathtt{f}^{\ell}};\ \overline{\mathtt{M}}\ \} \qquad \mathtt{t}^{\ell}\ \mathtt{m}(\overline{\mathtt{t}\ \mathtt{x}^{\ell}})\ \{\ \mathtt{return\ e;}\ \} \in \overline{\mathtt{M}}}{\begin{array}{c} mtype(\overline{\mathtt{CD}},\mathtt{C},\mathtt{m}) = \overline{\mathtt{T}} \to \mathtt{t} \\ mbody(\overline{\mathtt{CD}},\mathtt{C},\mathtt{m}) = (\overline{\mathtt{x}},\mathtt{e},\ell) \\ impl(\overline{\mathtt{CD}},\mathtt{C},\mathtt{m}) = \mathtt{C::m}\end{array}}$$

$$\frac{\overline{\mathtt{CD}}(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\ \overline{\mathtt{t}\ \mathtt{f}^{\ell}};\ \overline{\mathtt{M}}\ \} \qquad \mathtt{m}\ \text{not defined in}\ \overline{\mathtt{M}}}{\begin{array}{c} mtype(\overline{\mathtt{CD}},\mathtt{C},\mathtt{m}) = mtype(\overline{\mathtt{CD}},\mathtt{D},\mathtt{m}) \\ mbody(\overline{\mathtt{CD}},\mathtt{C},\mathtt{m}) = mbody(\overline{\mathtt{CD}},\mathtt{D},\mathtt{m}) \\ impl(\overline{\mathtt{CD}},\mathtt{C},\mathtt{m}) = impl(\overline{\mathtt{CD}},\mathtt{D},\mathtt{m})\end{array}}$$

$$\frac{\overline{\mathtt{CD}}(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\ \cdots\ \} \quad mtype(\overline{\mathtt{CD}},\mathtt{D},\mathtt{m}) = \overline{\mathtt{t'}} \to \mathtt{t'}\ \text{implies}\ \overline{\mathtt{t}} = \overline{\mathtt{t'}} \wedge \mathtt{t} = \mathtt{t'}}{can\text{-}declare(\overline{\mathtt{CD}},\mathtt{C},\mathtt{m},\overline{\mathtt{t}} \to \mathtt{t})}$$

The Retyping Function and the Transformation

$$\begin{array}{ll}
retype(\mathtt{(C_1,C_2)}^{\ell},\phi) & = \mathtt{(C_1,}\ \sqcup\phi(\ell)\mathtt{)} \\
 & \\
retype(\mathtt{x}^{\ell},\phi) & = \mathtt{x}^{\ell} \\
retype(\mathtt{new\ C}^{\ell}\mathtt{(}\overline{\mathtt{e}}\mathtt{)},\phi) & = \mathtt{new\ C}^{\ell}\mathtt{(}retype(\overline{\mathtt{e}},\phi)\mathtt{)} \\
retype(\mathtt{e.f}^{\ell},\phi) & = retype(\mathtt{e},\phi)\mathtt{.f}^{\ell} \\
retype(\mathtt{e.m}^{\ell}\mathtt{(}\overline{\mathtt{e}}\mathtt{)},\phi) & = retype(\mathtt{e},\phi)\mathtt{.m}^{\ell}\mathtt{(}retype(\overline{\mathtt{e}},\phi)\mathtt{)} \\
retype(\mathtt{(t)}^{\ell}\mathtt{e},\phi) & = \mathtt{(}retype(\mathtt{t}^{\ell},\phi)\mathtt{)}^{\ell} retype(\mathtt{e},\phi) \\
retype(\mathtt{dynnew}^{\ell},\phi) & = \mathtt{dynnew}^{\ell} \\
retype(\mathtt{e.[C::]}^{\ell}\mathtt{m(}\overline{\mathtt{e}}\mathtt{)},\phi) & = retype(\mathtt{e},\phi)\mathtt{.[C::]}^{\ell}\mathtt{m(}retype(\overline{\mathtt{e}},\phi)\mathtt{)} \\
 & \\
retype(\mathtt{t}^{\ell}\ \mathtt{m(}\overline{\mathtt{t}\ \mathtt{x}^{\ell}}\mathtt{)}\ \{\ \mathtt{return\ e;}\ \},\phi) & = retype(\mathtt{t}^{\ell},\phi)^{\ell}\ \mathtt{m(}retype(\overline{\mathtt{t}^{\ell}},\phi)\ \overline{\mathtt{x}^{\ell}}\mathtt{)} \\
 & \quad \{\ \mathtt{return}\ retype(\mathtt{e},\phi);\ \} \\
retype(\mathtt{class\ C_1\ extends\ C_2}\ \{\ \overline{\mathtt{t}\ \mathtt{f}^{\ell}};\ \overline{\mathtt{M}}\ \},\phi) & = \mathtt{class\ C_1\ extends\ C_2} \\
 & \quad \{\ retype(\overline{\mathtt{t}^{\ell}},\phi)\ \overline{\mathtt{f}^{\ell}};\ retype(\overline{\mathtt{M}},\phi)\ \} \\
\llbracket \mathtt{x}^{\ell} \rrbracket_{\overline{\mathtt{CD}},\phi} & = \mathtt{x}^{\ell} \\
\llbracket \mathtt{new\ C}^{\ell}\mathtt{(}\overline{\mathtt{e}}\mathtt{)} \rrbracket_{\overline{\mathtt{CD}},\phi} & = \mathtt{new\ C}^{\ell}\mathtt{(}\llbracket\overline{\mathtt{e}}\rrbracket_{\overline{\mathtt{CD}},\phi}\mathtt{)} \\
\llbracket \mathtt{e.f}^{\ell} \rrbracket_{\overline{\mathtt{CD}},\phi} & = \llbracket\mathtt{e}\rrbracket_{\overline{\mathtt{CD}},\phi}\mathtt{.f}^{\ell} \\
\llbracket \mathtt{e.m}^{\ell}\mathtt{(}\overline{\mathtt{e}}\mathtt{)} \rrbracket_{\overline{\mathtt{CD}},\phi} & = \llbracket\mathtt{e}\rrbracket_{\overline{\mathtt{CD}},\phi}\mathtt{.[C::]}^{\ell}\mathtt{m(}\llbracket\overline{\mathtt{e}}\rrbracket_{\overline{\mathtt{CD}},\phi}\mathtt{)} \\
 & \quad \text{if}\ \forall\mathtt{D} \in \phi(lab(\mathtt{e})) : impl(\overline{\mathtt{CD}},\mathtt{D},\mathtt{m}) = \mathtt{C::m} \\
\llbracket \mathtt{e.m}^{\ell}\mathtt{(}\overline{\mathtt{e}}\mathtt{)} \rrbracket_{\overline{\mathtt{CD}},\phi} & = \llbracket\mathtt{e}\rrbracket_{\overline{\mathtt{CD}},\phi}\mathtt{.m}^{\ell}\mathtt{(}\llbracket\overline{\mathtt{e}}\rrbracket_{\overline{\mathtt{CD}},\phi}\mathtt{)} \\
 & \quad \text{otherwise} \\
\llbracket \mathtt{(t)}^{\ell}\mathtt{e} \rrbracket_{\overline{\mathtt{CD}},\phi} & = \mathtt{(t)}^{\ell}\llbracket\mathtt{e}\rrbracket_{\overline{\mathtt{CD}},\phi} \\
\llbracket \mathtt{dynnew}^{\ell} \rrbracket_{\overline{\mathtt{CD}},\phi} & = \mathtt{dynnew}^{\ell} \\
\llbracket \mathtt{e.[C::]}^{\ell}\mathtt{m(}\overline{\mathtt{e}}\mathtt{)} \rrbracket_{\overline{\mathtt{CD}},\phi} & = \llbracket\mathtt{e}\rrbracket_{\overline{\mathtt{CD}},\phi}\mathtt{.[C::]}^{\ell}\mathtt{m(}\llbracket\overline{\mathtt{e}}\rrbracket_{\overline{\mathtt{CD}},\phi}\mathtt{)} \\
 & \\
\llbracket \mathtt{t}^{\ell}\ \mathtt{m(}\overline{\mathtt{t}\ \mathtt{x}^{\ell}}\mathtt{)}\ \{\ \mathtt{return\ e;}\ \} \rrbracket_{\overline{\mathtt{CD}},\phi} & = \mathtt{t}^{\ell}\ \mathtt{m(}\overline{\mathtt{t}\ \mathtt{x}^{\ell}}\mathtt{)}\ \{\ \mathtt{return}\ \llbracket\mathtt{e}\rrbracket_{\overline{\mathtt{CD}},\phi}\mathtt{;}\ \} \\
\llbracket \mathtt{class\ C_1\ extends\ C_2}\ \{\ \overline{\mathtt{t}\ \mathtt{f}^{\ell}};\ \overline{\mathtt{M}}\ \} \rrbracket_{\overline{\mathtt{CD}},\phi} & = \mathtt{class\ C_1\ extends\ C_2}\ \{\ \overline{\mathtt{t}\ \mathtt{f}^{\ell}};\ \llbracket\overline{\mathtt{M}}\rrbracket_{\overline{\mathtt{CD}},\phi}\ \}
\end{array}$$