Type-Safe Method Inlining

Neal Glew[a] Jens Palsberg[b]

[a]*Intel Corporation*
Santa Clara, CA 95054
aglew@acm.org

[b]*UCLA Computer Science Department*
4531K Boelter Hall, Los Angeles, CA 90095
palsberg@ucla.edu

Abstract

In a typed language such as Java, inlining of virtual methods does not always preserve typability. The best known solution to this problem is to insert type casts, which may hurt performance. This paper presents a solution that never hurts performance. The solution is based on a transformation that modifies static type annotations and changes some virtual calls into static calls, which can then be safely inlined. The transformation is parameterised by a flow analysis, and for any analysis that satisfies certain conditions, the transformation is correct and idempotent. The paper presents the transformation, the conditions on the flow analysis, and proves the correctness properties in the context of a variant of Featherweight Java.

Key words: Types, objects, inlining

1 Introduction

1.1 Background

Behavior-preserving program transformations can change the design or even the language of the programs they transform. They are key to several parts of the software engineering process. Compilers, for example, transform programs for efficiency and to translate from high-level languages to machine code. Software engineers also transform programs to improve the design, maintain the program, or evolve the program towards new goals. In the context of object-oriented software engineering, the whole area of refactoring [1] employs program transformations to achieve its goals of evolving software and in particular the design of the software. As such efforts become increasingly ambitious, so does the amount of detail that must be attended to and the importance of doing the transformations correctly. Hence there is an increasing interest in tool support for program transformation. Such tool support is challenging because of new languages and new language features.

Some of today's popular programming languages, including Java [2] and C++ [3], have static type systems. Java also has a bytecode language with a notion of type soundness, based on the concept of bytecode verification [4]. Transformations of programs in these languages must produce programs in these languages themselves, and in particular produce programs that still type check—a property called typability preservation. In a similar sense, a number of recent compilers use typed intermediate languages (*e.g.*, [5–8]) to obtain debugging and optimisation benefits [5,9]. Such compilers also require transformations that are typability preserving when translating from one typed representation to another.

One recent example of a typability-preserving program transformation in the context of software engineering was given by Tip, Kiezun, and Baumer [10]. They provided tool support for a transformation known as *Extract Interface/Superclass* for redirecting the access to a class via a newly created interface. This refactoring involves the updating of the types of variables, method parameters, method return types, and field types to make use of the newly added interface.

1.2 The Problem

This paper is concerned with a particular transformation, namely that of inlining of dynamic method calls in a statically-typed object-oriented language. Method inlining is standard in industrial-strength Java virtual machines, see for example [11]. Method inlining is also useful for software engineering and program maintenance; it is a standard refactoring operation that is supported by interactive development environments such as IntelliJ (see `http://www.intellij.com`). In Java, all method calls are dynamic; they are also known as virtual calls. For comparison, C++ has both dynamic and static calls. While there has been substantial previous work on method inlining (*e.g.*, [12,13]), the known approaches are either for an untyped language, or have to rely on adding type casts or extra types. For example, consider the following well-typed Java program. Note here that from the perspective of preserving type correctness, there is no major difference between working with Java source code and working with Java bytecode.

```
class B {                           // a code snippet:
  B m() { return this; }            B x = new C();
}                                   x = x.m();
                                    x = ((B)new C()).m();
class C extends B {
  C f;
  B m() {
    return this.f;
  }
}
```

Both of the method calls `x.m()` and `((B)new C()).m()` have a unique target method that is a small code fragment, so it makes sense to inline these calls.

In both cases, a compiler could inline by taking the body of `m` and replacing `this` with the actual receiver expression to get:

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

These two assignments do not type check. The reason is that while `this` in class `C` has static type `C`, both `x` and `(B)new C()` have static type `B`. Hence, both `x.f` and `((B)new C()).f` will yield the compile-time error that there is no `f`-field in either `x` or `(B)new C()`.

The problem can be solved by inserting type casts. In their Java compiler, Wright et al. [7] insert type casts (in the form of a typecase expression) of `this` in all translated method bodies. Applying this idea to our example program produces the following declaration for method `m` in class `C`:

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

After inlining, the two assignments type check:

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

A different approach was taken by Gagnon et al. [14] who first compile Java to a representation of Java bytecode in which variables do not have types, then do the optimizations on that representation, and finally infer types to regain static type annotations. Gagnon et al. use a style of type inference that combines a flow-based style [15] with the types of methods that came from the Java bytecode. Their results show that this works well for a substantial suite of benchmark programs. In general, however, their algorithm for type inference may fail, and in such cases they revert to inserting type casts.

A related approach, which does not require type casts at all, is to add new types to the program. Knoblock and Rehof [16] demonstrated how to add types in a way such that type inference will succeed for all verifiable Java bytecode programs.

In general, inserting type casts may hurt performance, and adding new types may not be acceptable. Since these type casts and extra types are not added in the untyped setting, they are there just for the purposes of satisfying the type system. It is intellectually unsatisfying that we cannot just use the untyped techniques. Until now, it has remained an open problem to devise a scheme for supporting typability-preserving method inlining in a way that does not require the insertion of type casts or extra types. This paper solves the problem.

1.3 Our Approach

The core of the problem is an instance of what we call *type rot*. Perfectly fine type annotations somehow "rot" during a step of method inlining. Before the transformation, the program type checks, but after the transformation, the *same* type annotations are suddenly no good. This observation leads us to the following insight:

> **Insight 1:** For method inlining, transforming statements and expressions is insufficient; we must also transform the type annotations and type casts.

For the code snippet in Section 1.2, the type of `x` is `B`, even though the more precise type `C` could also be used. Similarly, the cast to `B` could as well be a cast to `C`. Thus, we can transform the types in a way that preserves well-typedness:

```
C x = new C();          // the type of x has been changed to C
x = x.m();
x = ((C)new C()).m();   // the type cast has been changed to C
```

Inlining then produces the following well-typed code snippet:

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

Notice that for the example, the type transformation does not change the behavior of the program, and, hence, performance is not affected.

To ensure that our type transformation preserves well-typedness, it is designed carefully in the following way. Each type annotation and type case is either left unchanged or is changed to a subtype. For the example, such type transformation is sufficient to enable type-safe method inlining. However, this is not always the case. In Section 5, we present an example with a method call that has a unique receiver but where no type transformation can enable type-safe method inlining.

The centerpiece of our approach is showing that type transformation can be automated and can enable a large number of inlinings.

1.4 Our Result

We present an approach to typability-preserving method inlining that never hurts performance and does not require the insertion of type casts or new types. Our approach has three components:

(1) **Type transformation:** We change some type annotations and type casts to be more precise.
(2) **Devirtualisation:** If a dynamic call has a unique target, then it can be devirtualised, that is, changed to a static call.
(3) **Inlining:** We inline the static calls.

Notice that we view devirtualisation and inlining as two separate components. As in previous work, a dynamic method call $\mathtt{e.m(e_1,\ldots,e_n)}$ can be transformed to a static call $\mathtt{e.D::m(e_1,\ldots,e_n)}$ if all the objects that `e` could evaluate to are instances of classes that inherit `m` from a fixed class `D`. (The expression $\mathtt{e.D::m(e_1,\ldots,e_n)}$ invokes `D`'s version of `m` on `e` with arguments $\mathtt{e}_1$ through $\mathtt{e}_n$.) The static call $\mathtt{e.D::m(e_1,\ \ldots,\ e_n)}$ can be inlined to $\mathtt{e}'\{\mathtt{this}, \mathtt{x}_1, \ldots, \mathtt{x}_n := \mathtt{e}, \mathtt{e}_1, \ldots, \mathtt{e}_n\}$ where `D` has method `m` with body $\mathtt{e}'$ and parameters $\mathtt{x}_1$ through $\mathtt{x}_n$. Inlining of a static call is nothing other than applying a nonstandard reduction rule at compile time, and it is straightforward to show that the rule is typability preserving.

So, the main difficulty is to do type transformation and devirtualisation in a typability-preserving manner. For both of them, a compiler needs information that can drive the transformations. In the case of devirtualisation, a standard approach is *flow-directed* devirtualisation. The idea is to use a static program analysis, known as flow analysis, which approximates the results of evaluating expressions. For each expression, it determines a set of classes such that every possible result of evaluating the expression is an instance of one of those classes. Based on such information, a compiler can easily determine whether a dynamic call has a unique target.

The set of classes computed by a flow analysis is easily transformed into a Java type. In particular, the least upper bound of the set, if it exists, can replace the old type annotation. If a least upper bound does not exists, such as for multiple extensions of interfaces, we can fall back to the old static type. We call this *flow-directed type transformation*.

For our example program in Section 1.2, the best flow set for both receiver expressions in the program is {`C`}, and the least upper bound for this set is `C`. Therefore the program transforms into the one shown in Section 1.3; it type checks.

Given that we can do both type transformation and devirtualisation in a flow-directed manner, we are led to our second key insight:

> **Insight 2:** Type transformation and devirtualisation should be done together.

The idea is to do a single flow analysis and then do both of the type transformation and the devirtualisation based on the *same* flow information. While, in theory, one can imagine the use of two flow analyses, it makes sense for a compiler to do just one.

Not all flow analyses enable type-safe method inlining in the style above. However, we give sufficient conditions on a flow analysis for ensuring correctness. We will present the transformation, the conditions on the flow analysis, and prove the correctness properties; all in the context of a variant of Featherweight Java. It is straightforward to extend our approach to full Java. (Students at Purdue have implemented our ideas for the full Java Virtual Machine.)

Because the flow analysis is used for flow-directed type transformation, it is crucial to align the flow analysis with the type system. There are several aspects of Java's type system that lead to unusual conditions on the flow analysis. One example is Java's lack of a bottom type, leading to a nonemptyness condition on flow sets. Another example is that the Java type system allows the use of subtyping in some places but not in others. One of the insights from previous work on aligning flow analysis and type systems [17–20,16] is that subset constraints correspond to subtyping, while equality constraints correspond to "no nontrivial subtyping," that is, types are related only if they are equal. The consequence is that a flow analyses must satisfy subset constraints in some places and equality constraints in others (see [21,22] for examples of subset constraints and [23] for an example of equality constraints and another example of the mixed use of subset and equality constraints).

Note that our transformation is based on a flow analysis which is a whole-program analysis. Hence, our transformation is a whole-program transformation. By making suitable conservative assumptions it could be used to transform separate program fragments. How effective this might be is beyond the scope of this paper. More recent work [24] has extended our ideas to a dynamic class loading environment with a just-in-time compiler. Note also that our flow analyses are context insensitive and we leave context-sensitive flow analyses to future work.

The following section presents our variant of Featherweight Java, Section 3 presents the constraints flow analyses must satisfy, Section 4 presents the program transformation, and Section 5 discusses some examples. The proofs of the correctness theorems are presented in three appendices.

2 The Language

We formalise our results in Featherweight Java [25] (FJ) extended with a static call construct, a language we call FJS. The language and its presentation follow the original FJ paper as closely as possible.

As in FJ, an FJS program is a list of class definitions and an expression to be evaluated. Each class definition is in a stylised form. Every class extends another; top-level classes extend `Object`. Every class has exactly one constructor. This constructor has one parameter for each of the fields of the class, with the same names and in the same order. It first calls the superclass constructor with the parameters that correspond to the superclass's fields. Then it uses the remaining parameters to initialise the fields declared in the class. Constructors are the only place where `super` or `=` appear in an FJS program. The receiver of a field access or method call is always explicit; `this` is used to refer to an object's fields and methods. FJS is functional, so a method body consists just of a return statement with an expression and there is no `void` type. There are just six forms of expressions: variables, field access, object constructors, dynamic casts, dynamic method call, and static method call. Although FJS does not have `super`, static method call can be used to call a superclass's methods. The remainder of this section formalises the language.

2.1 Syntax and Semantics

The syntax of FJS is:

$$\begin{array}{lll} \mathtt{P} & ::= & (\overline{\mathtt{CD}},\mathtt{e}) \\ \mathtt{CD} & ::= & \mathtt{class\ C\ extends\ C\ \{}\overline{\mathtt{C}}\ \overline{\mathtt{f}^{\ell}}\mathtt{;\ K\ }\overline{\mathtt{M}}\mathtt{\}} \end{array}$$

$$\begin{array}{lll}
\texttt{K} & ::= & \texttt{C}(\overline{\texttt{C}}\ \overline{\texttt{f}})\ \{\texttt{super}(\overline{\texttt{f}});\ \texttt{this}.\overline{\texttt{f}} = \overline{\texttt{f}};\} \\
\texttt{M} & ::= & \texttt{C m}(\overline{\texttt{C}}\ \overline{\texttt{x}^\ell})\ \{\texttt{return}^\ell\texttt{e};\} \\
\texttt{e} & ::= & \texttt{x}^\ell \mid \texttt{e.f}^\ell \mid \texttt{new}^\ell\ \texttt{C}(\overline{\texttt{e}}) \mid (\texttt{C})^\ell\texttt{e} \mid \texttt{e.m}(\overline{\texttt{e}})^\ell \mid \texttt{e.C::m}(\overline{\texttt{e}})^\ell
\end{array}$$

The metavariables A, B, C, D, and E range over class names; f and g range over field names; m ranges over method names; x ranges over variables; d and e range over expressions; M ranges over method definitions; K ranges over constructors; CD ranges over class definitions; and P ranges over programs. Object is a class name, but no program may give it a definition; this is a variable, but no program may use it as a parameter. The over bar notation denotes sequences, so $\overline{\texttt{f}}$ abbreviates $\texttt{f}_1, \ \ldots, \ \texttt{f}_n$. This notation also denotes pairs of sequences in an obvious way—$\overline{\texttt{C}}\ \overline{\texttt{f}^\ell}$ abbreviates $\texttt{C}_1\ \texttt{f}_1^{\ell_1}, \ \ldots,$ $\texttt{C}_n\ \texttt{f}_n^{\ell_n}$, $\overline{\texttt{C}}\ \overline{\texttt{f}^\ell};$ abbreviates $\texttt{C}_1\ \texttt{f}_1^{\ell_1};\ \cdots;\ \texttt{C}_n\ \texttt{f}_n^{\ell_n};$, and $\texttt{this}.\overline{\texttt{f}}=\overline{\texttt{f}}$ abbreviates $\texttt{this.f}_1\texttt{=f}_1;\ \cdots;\ \texttt{this.f}_n\texttt{=f}_n$. The empty sequence is •, and comma concatenates sequences. Sequences of class definitions, field declarations, method definitions, and parameter declarations may not contain duplicate names. We abuse notation and consider a sequence of class definitions to also be a mapping from class names to class definitions, and write $\overline{\texttt{CD}}(\texttt{C})$ to mean the definition of C under the map corresponding to $\overline{\texttt{CD}}$. Any class name C except Object appearing in a program must be given a definition by that program, and the extends clauses of a program must be acyclic.

Class definition $\texttt{class C extends D}\ \{\overline{\texttt{E}}\ \overline{\texttt{f}^\ell};\ \texttt{K}\ \overline{\texttt{M}}\}$ declares class C to be a subclass of D. In addition to the fields of its superclass, C has fields $\overline{\texttt{f}^\ell}$ of types $\overline{\texttt{E}}$. K is the constructor for the class, and it has the stylised form described above. $\overline{\texttt{M}}$ are the methods declared by C, they may be new methods or may override those of D. C also inherits all methods of D that it does not override. Method declaration $\texttt{C m}(\overline{\texttt{D}}\ \overline{\texttt{x}^\ell})\ \{\texttt{return}^\ell\texttt{e};\}$ declares a method m with return type C, with parameters $\overline{\texttt{x}^\ell}$ of types $\overline{\texttt{D}}$, and that when invoked evaluates expression e and returns it as the result of the call.

As mentioned above, there are six forms of expression: variables x, field selection $\texttt{e.f}^\ell$, object constructors $\texttt{new}^\ell\ \texttt{C}(\overline{\texttt{e}})$, casts $(\texttt{C})^\ell\texttt{e}$, dynamic method calls $\texttt{e.m}(\overline{\texttt{d}})^\ell$, and static method calls $\texttt{e.C::m}(\overline{\texttt{d}})^\ell$. The latter invokes C's version of method m on object e, which should be in C or one of its subclasses.

Metavariable ℓ ranges over a set of labels. Such a label is used, for example, in $\texttt{C f}^\ell$ to label the field f. Notice that there is a label associated with all expressions, fields, method returns, and formal arguments; these labels are assumed to be unique. For a program P, *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. Furthermore, the label on any variable occurrence must be the same as the label on its declaration, and any two occurrences of this in a class must have the same label. Any well-typed program can easily

Field Lookup:

$$\frac{}{\mathit{fields}(\overline{\mathtt{CD}}, \mathtt{Object}) = \bullet} \tag{1}$$

$$\frac{\overline{\mathtt{CD}}(\mathtt{C}) = \mathtt{class\ C\ extends\ C_0\ \{\overline{D_2}\ \overline{f^{\ell}};\ K\ \overline{M}\}} \qquad \mathit{fields}(\overline{\mathtt{CD}}, \mathtt{C_0}) = \overline{\mathtt{D}}_1\ \overline{\mathtt{g}}}{\mathit{fields}(\overline{\mathtt{CD}}, \mathtt{C}) = \overline{\mathtt{D}}_1\ \overline{\mathtt{g}}, \overline{\mathtt{D}}_2\ \overline{\mathtt{f}}} \tag{2}$$

Method Type Lookup:

$$\frac{\begin{array}{c}\overline{\mathtt{CD}}(\mathtt{C}) = \mathtt{class\ C\ extends\ C_0\ \{\overline{D}\ \overline{f^{\ell_1}};\ K\ \overline{M}\}} \\ \mathtt{B_0\ m(\overline{B}\ \overline{x^{\ell_2}})\ \{return^{\ell}e;\}} \in \overline{\mathtt{M}}\end{array}}{\mathit{mtype}(\overline{\mathtt{CD}}, \mathtt{C}, \mathtt{m}) = \overline{\mathtt{B}} \to \mathtt{B_0}} \tag{3}$$

$$\frac{\overline{\mathtt{CD}}(\mathtt{C}) = \mathtt{class\ C\ extends\ C_0\ \{\overline{D}\ \overline{f^{\ell}};\ K\ \overline{M}\}} \qquad \mathtt{m} \text{ not defined in } \overline{\mathtt{M}}}{\mathit{mtype}(\overline{\mathtt{CD}}, \mathtt{C}, \mathtt{m}) = \mathit{mtype}(\overline{\mathtt{CD}}, \mathtt{C_0}, \mathtt{m})} \tag{4}$$

Method Body Lookup:

$$\frac{\begin{array}{c}\overline{\mathtt{CD}}(\mathtt{C}) = \mathtt{class\ C\ extends\ C_0\ \{\overline{D}\ \overline{f^{\ell_1}};\ K\ \overline{M}\}} \\ \mathtt{B_0\ m(\overline{B}\ \overline{x^{\ell_2}})\ \{return^{\ell}e;\}} \in \overline{\mathtt{M}}\end{array}}{\mathit{mbody}(\overline{\mathtt{CD}}, \mathtt{C}, \mathtt{m}) = (\ell, \overline{\mathtt{x}}, \mathtt{e})} \tag{5}$$

$$\frac{\overline{\mathtt{CD}}(\mathtt{C}) = \mathtt{class\ C\ extends\ C_0\ \{\overline{D}\ \overline{f^{\ell_1}};\ K\ \overline{M}\}} \qquad \mathtt{m} \text{ not defined in } \overline{\mathtt{M}}}{\mathit{mbody}(\overline{\mathtt{CD}}, \mathtt{C}, \mathtt{m}) = \mathit{mbody}(\overline{\mathtt{CD}}, \mathtt{C_0}, \mathtt{m})} \tag{6}$$

Class of Method Lookup:

$$\frac{\begin{array}{c}\overline{\mathtt{CD}}(\mathtt{C}) = \mathtt{class\ C\ extends\ C_0\ \{\overline{D}\ \overline{f^{\ell_1}};\ K\ \overline{M}\}} \\ \mathtt{B_0\ m(\overline{B}\ \overline{x^{\ell_2}})\ \{return^{\ell}e;\}} \in \overline{\mathtt{M}}\end{array}}{\mathit{impl}(\overline{\mathtt{CD}}, \mathtt{C}, \mathtt{m}) = \mathtt{C::m}} \tag{7}$$

$$\frac{\overline{\mathtt{CD}}(\mathtt{C}) = \mathtt{class\ C\ extends\ C_0\ \{\overline{D}\ \overline{f^{\ell_1}};\ K\ \overline{M}\}} \qquad \mathtt{m} \text{ not defined in } \overline{\mathtt{M}}}{\mathit{impl}(\overline{\mathtt{CD}}, \mathtt{C}, \mathtt{m}) = \mathit{impl}(\overline{\mathtt{CD}}, \mathtt{C_0}, \mathtt{m})} \tag{8}$$

Valid Method Overriding:

$$\frac{\mathit{mtype}(\overline{\mathtt{CD}}, \mathtt{D}, \mathtt{m}) = \overline{\mathtt{E}} \to \mathtt{E_0} \text{ implies } \overline{\mathtt{C}} = \overline{\mathtt{E}} \text{ and } \mathtt{C_0} = \mathtt{E_0}}{\mathit{can\text{-}declare}(\overline{\mathtt{CD}}, \mathtt{D}, \mathtt{m}, \overline{\mathtt{C}} \to \mathtt{C_0})} \tag{9}$$

Fig. 1. Auxiliary Definitions

be transformed to satisfy these conditions. Function *lab* maps an expression, a field name, or an argument name to its label.

Some auxiliary definitions that are used in the rest of the paper appear in Figure 1. Unlike FJ, we do not make the list of class declarations global, but have them appear explicitly as parameters to functions, predicates, and rules. Function $\mathit{fields}(\overline{\mathtt{CD}}, \mathtt{C})$ returns a list of C's fields and their types; $\mathit{mtype}(\overline{\mathtt{CD}}, \mathtt{C}, \mathtt{m})$

$$\frac{\mathit{fields}(\overline{\mathtt{CD}},\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}}}{(\overline{\mathtt{CD}},\mathtt{X}\langle\mathtt{new}^{\ell_1}\ \mathtt{C}(\overline{\mathtt{e}})\mathtt{.f}_i{}^{\ell_2}\rangle) \mapsto (\overline{\mathtt{CD}},\mathtt{X}\langle\mathtt{e}_i\rangle)} \tag{10}$$

$$\frac{\overline{\mathtt{CD}} \vdash \mathtt{C} \mathrel{\mathtt{<:}} \mathtt{D}}{(\overline{\mathtt{CD}},\mathtt{X}\langle(\mathtt{D})^{\ell_1}\mathtt{new}^{\ell_2}\ \mathtt{C}(\overline{\mathtt{e}})\rangle) \mapsto (\overline{\mathtt{CD}},\mathtt{X}\langle\mathtt{new}^{\ell_2}\ \mathtt{C}(\overline{\mathtt{e}})\rangle)} \tag{11}$$

$$\frac{\mathit{mbody}(\overline{\mathtt{CD}},\mathtt{C},\mathtt{m}) = (\ell,\overline{\mathtt{x}},\mathtt{e}_0)}{(\overline{\mathtt{CD}},\mathtt{X}\langle\mathtt{new}^{\ell_1}\ \mathtt{C}(\overline{\mathtt{e}})\mathtt{.m}(\overline{\mathtt{d}})^{\ell_2}\rangle) \mapsto (\overline{\mathtt{CD}},\mathtt{X}\langle\mathtt{e}_0\{\mathtt{this},\overline{\mathtt{x}} := \mathtt{new}^{\ell_1}\ \mathtt{C}(\overline{\mathtt{e}}),\overline{\mathtt{d}}\}\rangle)} \tag{12}$$

$$\frac{\mathit{mbody}(\overline{\mathtt{CD}},\mathtt{D},\mathtt{m}) = (\ell,\overline{\mathtt{x}},\mathtt{e}_0)}{(\overline{\mathtt{CD}},\mathtt{X}\langle\mathtt{new}^{\ell_1}\ \mathtt{C}(\overline{\mathtt{e}})\mathtt{.D::m}(\overline{\mathtt{d}})^{\ell_2}\rangle) \mapsto (\overline{\mathtt{CD}},\mathtt{X}\langle\mathtt{e}_0\{\mathtt{this},\overline{\mathtt{x}} := \mathtt{new}^{\ell_1}\ \mathtt{C}(\overline{\mathtt{e}}),\overline{\mathtt{d}}\}\rangle)} \tag{13}$$

Fig. 2. Operational Semantics

returns the type of method m in class C, this type has the form $\overline{\mathtt{D}} \to \mathtt{D}_0$ where $\mathtt{D}_0$ is the return type and $\overline{\mathtt{D}}$ are the argument types; $\mathit{mbody}(\overline{\mathtt{CD}},\mathtt{C},\mathtt{m})$ returns the body of method m in class C, this has the form $(\ell,\overline{\mathtt{x}},\mathtt{e})$ where ℓ is the label of the return statement, e is the expression to evaluate, and $\overline{\mathtt{x}}$ are the parameter names; $\mathit{impl}(\overline{\mathtt{CD}},\mathtt{C},\mathtt{m})$ returns the class from which class C inherits method m (this might be C itself if C declares m), this has the form D::m where D is the class. Predicate $\mathit{can\text{-}declare}(\overline{\mathtt{CD}},\mathtt{D},\mathtt{m},\overline{\mathtt{C}} \to \mathtt{C}_0)$ is true when method m of type $\overline{\mathtt{C}} \to \mathtt{C}_0$ 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 (the definition of acceptable flow would change slightly). Notice that the definition of $\mathit{can\text{-}declare}(\overline{\mathtt{CD}},\mathtt{D},\mathtt{m},\overline{\mathtt{C}} \to \mathtt{C}_0)$ uses an implication rather than, say, a conjunction. This is because the definition captures *both* the case where no method m was declared in D or a superclass of D *and* the case where a method m was indeed declared in D or a superclass of D.

The operational semantics of the language appear in Figure 2. Metavariable X ranges over evaluation contexts, which are expressions with exactly one hole; $\mathtt{X}\langle\mathtt{e}\rangle$ denotes the expression formed by replacing the hole in X by the expression e. Unlike FJ, 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 $\mathtt{new}^{\ell}\ \mathtt{C}(\overline{\mathtt{e}})$. Field access reduces to the appropriate element of $\overline{\mathtt{e}}$. The cast $(\mathtt{C})^{\ell_1}\mathtt{new}^{\ell_2}\ \mathtt{D}(\overline{\mathtt{e}})$ reduces to the object $\mathtt{new}^{\ell_2}\ \mathtt{D}(\overline{\mathtt{e}})$ if D is a subclass of C. If D is not a subclass of C then the cast is irreducible representing that the cast fails as a checked run-time error. The method call $\mathtt{new}^{\ell_1}\ \mathtt{C}(\overline{\mathtt{e}})\mathtt{.m}(\overline{\mathtt{d}})^{\ell_2}$ reduces to the method body $\mathtt{e}_0$ with the actual parameters $\overline{\mathtt{d}}$ substituted for the formal parameters $\overline{\mathtt{x}}$ and the object $\mathtt{new}^{\ell_1}\ \mathtt{C}(\overline{\mathtt{e}})$ substituted for this. The method body and formal parameters are obtained by looking up the method m in class C, $\mathit{mbody}(\overline{\mathtt{CD}},\mathtt{C},\mathtt{m}) = (\ell,\overline{\mathtt{x}},\mathtt{e}_0)$. Static method call $\texttt{new}^{\ell_1}\ \texttt{C}(\overline{\texttt{e}})\texttt{.D::m}(\overline{\texttt{d}})^{\ell}$ 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 call can be implemented as a direct call rather than an indirect call through a method table.

An irreducible expression is *stuck* if it is of the form $\texttt{X}\langle\texttt{e.f}^{\ell}\rangle$, $\texttt{X}\langle\texttt{e.m}(\overline{\texttt{d}})^{\ell}\rangle$, or $\texttt{X}\langle\texttt{e.D::m}(\overline{\texttt{d}})^{\ell}\rangle$. The type system prevents stuck expressions from occurring during execution of a program (see Theorem 1). Irreducible expressions that are not stuck are of the form $\texttt{v::=new}^{\ell}\ \texttt{C}(\overline{\texttt{v}})$ or $\texttt{X}\langle(\texttt{C})^{\ell_1}\texttt{new}^{\ell_2}\ \texttt{D}(\overline{\texttt{e}})\rangle$ where D is not a subclass of C; the former represents normal termination with a fully evaluated object, the latter represents a failed cast.

2.2 Type System

The type system consists of the following judgements:

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

A typing context Γ has the form $\overline{\texttt{x}}\texttt{:}\overline{\texttt{C}}$ 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 $\overline{\texttt{CD}}; \Gamma \vdash \overline{\texttt{e}} \in \overline{\texttt{C}}$ abbreviates $\overline{\texttt{CD}}; \Gamma \vdash \texttt{e}_1 \in \texttt{C}_1, \ldots, \overline{\texttt{CD}}; \Gamma \vdash \texttt{e}_n \in \texttt{C}_n$.

The rules for constructors and method call check that each actual parameter has a subtype of the corresponding formal parameter. The typing rule for dynamic method call looks up the type of the method in the class of the receiver. The typing rule for static method call $\texttt{e.D::m}(\overline{\texttt{d}})^{\ell}$ requires that e has some subtype of D and looks up the type of the method in D. As in FJ, the typing rules allow *stupid casts*, such as $(\texttt{C})^{\ell_1}\texttt{new}^{\ell_2}\ \texttt{D}(\overline{\texttt{e}})$ where D is not a subclass of C and the cast will always fail. Allowing stupid casts is needed to prove type preservation. Unlike FJ, FJS has only one rule for casts, which just requires the expression being cast to have some type. This rule is equivalent to FJ's three rules except that it does not issue stupid-cast warnings. The type system is sound, that is, well-typed programs never get stuck. This fact is stated in the following theorem, which can be proved by standard

Subtyping:

$$\frac{}{\overline{\mathtt{CD}} \vdash \mathtt{C} \mathrel{\mathtt{<:}} \mathtt{C}} \tag{14}$$

$$\frac{\overline{\mathtt{CD}} \vdash \mathtt{C} \mathrel{\mathtt{<:}} \mathtt{D} \qquad \overline{\mathtt{CD}} \vdash \mathtt{D} \mathrel{\mathtt{<:}} \mathtt{E}}{\overline{\mathtt{CD}} \vdash \mathtt{C} \mathrel{\mathtt{<:}} \mathtt{E}} \tag{15}$$

$$\frac{\overline{\mathtt{CD}}(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{...\}}}{\overline{\mathtt{CD}} \vdash \mathtt{C} \mathrel{\mathtt{<:}} \mathtt{D}} \tag{16}$$

Expression Typing:

$$\frac{}{\overline{\mathtt{CD}}; \Gamma \vdash \mathtt{x}^{\ell} \in \Gamma(\mathtt{x})} \tag{17}$$

$$\frac{\overline{\mathtt{CD}}; \Gamma \vdash \mathtt{e}_0 \in \mathtt{C}_0 \qquad \mathit{fields}(\overline{\mathtt{CD}}, \mathtt{C}_0) = \overline{\mathtt{C}}\ \ \overline{\mathtt{f}}}{\overline{\mathtt{CD}}; \Gamma \vdash \mathtt{e}_0.\mathtt{f}_i^{\ell} \in \mathtt{C}_i} \tag{18}$$

$$\frac{\mathit{fields}(\overline{\mathtt{CD}}, \mathtt{C}) = \overline{\mathtt{D}}\ \ \overline{\mathtt{f}} \qquad \overline{\mathtt{CD}}; \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{E}} \qquad \overline{\mathtt{CD}} \vdash \overline{\mathtt{E}} \mathrel{\mathtt{<:}} \overline{\mathtt{D}}}{\overline{\mathtt{CD}}; \Gamma \vdash \mathtt{new}^{\ell}\ \ \mathtt{C}(\overline{\mathtt{e}}) \in \mathtt{C}} \tag{19}$$

$$\frac{\overline{\mathtt{CD}}; \Gamma \vdash \mathtt{e}_0 \in \mathtt{D}}{\overline{\mathtt{CD}}; \Gamma \vdash (\mathtt{C})^{\ell}\mathtt{e}_0 \in \mathtt{C}} \tag{20}$$

$$\frac{\overline{\mathtt{CD}}; \Gamma \vdash \mathtt{e}_0 \in \mathtt{C}_0 \qquad \mathit{mtype}(\overline{\mathtt{CD}}, \mathtt{C}_0, \mathtt{m}) = \overline{\mathtt{B}} \to \mathtt{C} \qquad \overline{\mathtt{CD}}; \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{E}} \qquad \overline{\mathtt{CD}} \vdash \overline{\mathtt{E}} \mathrel{\mathtt{<:}} \overline{\mathtt{B}}}{\overline{\mathtt{CD}}; \Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}})^{\ell} \in \mathtt{C}} \tag{21}$$

$$\frac{\begin{array}{l} \overline{\mathtt{CD}}; \Gamma \vdash \mathtt{e}_0 \in \mathtt{C}_0 \qquad \overline{\mathtt{CD}} \vdash \mathtt{C}_0 \mathrel{\mathtt{<:}} \mathtt{D} \\ \mathit{mtype}(\overline{\mathtt{CD}}, \mathtt{D}, \mathtt{m}) = \overline{\mathtt{B}} \to \mathtt{C} \\ \overline{\mathtt{CD}}; \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{E}} \qquad \overline{\mathtt{CD}} \vdash \overline{\mathtt{E}} \mathrel{\mathtt{<:}} \overline{\mathtt{B}} \end{array}}{\overline{\mathtt{CD}}; \Gamma \vdash \mathtt{e}_0.\mathtt{D}::\mathtt{m}(\overline{\mathtt{e}})^{\ell} \in \mathtt{C}} \tag{22}$$

Method Typing:

$$\frac{\begin{array}{l} \overline{\mathtt{CD}}; \mathtt{this} : \mathtt{D}, \overline{\mathtt{x}} : \overline{\mathtt{C}} \vdash \mathtt{e}_0 \in \mathtt{E}_0 \qquad \overline{\mathtt{CD}} \vdash \mathtt{E}_0 \mathrel{\mathtt{<:}} \mathtt{C}_0 \\ \overline{\mathtt{CD}}(\mathtt{C}) = \mathtt{class\ D\ extends\ D}_0\ \mathtt{\{...\}} \qquad \mathit{can\text{-}declare}(\overline{\mathtt{CD}}, \mathtt{D}_0, \mathtt{m}, \overline{\mathtt{C}} \to \mathtt{C}_0) \end{array}}{\overline{\mathtt{CD}} \vdash \mathtt{C}_0\ \ \mathtt{m}(\overline{\mathtt{C}}\ \ \overline{\mathtt{x}^{\ell}})\ \ \mathtt{\{return}^{\ell}\mathtt{e}_0\mathtt{;\}}\ \text{OK in } \mathtt{D}} \tag{23}$$

Class Typing:

$$\frac{\begin{array}{l} \overline{\mathtt{CD}} \vdash \overline{\mathtt{M}}\ \text{OK in } \mathtt{C} \\ \mathit{fields}(\overline{\mathtt{CD}}, \mathtt{C}_0) = \overline{\mathtt{D}}_1\ \ \overline{\mathtt{g}} \\ \mathtt{K} = \mathtt{C}(\overline{\mathtt{D}}_1\ \overline{\mathtt{g}},\ \overline{\mathtt{D}}_2\ \overline{\mathtt{f}})\ \mathtt{\{super(}\overline{\mathtt{g}}\mathtt{);\ this.}\overline{\mathtt{f}}\mathtt{=}\overline{\mathtt{f}}\mathtt{;\}} \end{array}}{\overline{\mathtt{CD}} \vdash \mathtt{class\ C\ extends\ C}_0\ \mathtt{\{}\overline{\mathtt{D}}_2\ \ \overline{\mathtt{f}^{\ell}}\mathtt{;\ K}\ \ \overline{\mathtt{M}}\mathtt{\}}\ \text{OK}} \tag{24}$$

Program Typing:

$$\frac{\overline{\mathtt{CD}} \vdash \overline{\mathtt{CD}}\ \text{OK} \qquad \overline{\mathtt{CD}}; \bullet \vdash \mathtt{e} \in \mathtt{C}}{\vdash (\overline{\mathtt{CD}}, \mathtt{e}) \in \mathtt{C}} \tag{25}$$

Fig. 3. Typing Rules

methods [26,27,25].

Theorem 1 (Type Soundness) *If* $\vdash$ `P` $\in$ `C` *then* `P` *does not reduce to a program with a stuck expression.*

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, which can be the label of a field, method parameter, method return, or expression appearing in `P`, there is a uniquely determined static type for the program point labeled ℓ, written $\textit{static-type}(\ell, \texttt{P})$.

3 Flow Analysis

A flow analysis approximates the results of evaluating expressions. In our setting, flow information for an expression is a set of classes such that the expression will evaluate to an instance of one of those classes.

For a program `P`, $\textit{classes}(\texttt{P})$ denotes the set of class names declared in `P`, $\textit{flow}(\texttt{P})$ is the powerset of $\textit{classes}(\texttt{P})$; elements of $\textit{flow}(\texttt{P})$ are called flows. The set $\textit{subclasses}(\texttt{P}, \texttt{C})$ is the set of subclasses of `C` (including `C`). Flow information for `P` is a member of $\textit{flow-information}(\texttt{P}) = \textit{labels}(\texttt{P}) \rightarrow \textit{flow}(\texttt{P})$—it associates a flow with each expression, field, method parameter, and method return. Metavariables S and T range over $\textit{flow}(\texttt{P})$, φ ranges over $\textit{flow-information}(\texttt{P})$. We order $\textit{flow-information}(\texttt{P})$ such that $\varphi_1 \leq \varphi_2$ if and only if $\varphi_1(\ell) \subseteq \varphi_2(\ell)$ for every $\ell \in \textit{labels}(\texttt{P})$. In $\textit{flow-information}(\texttt{P})$, the least element is $\lambda\ell.\emptyset$ and the greatest element is $\lambda\ell.\textit{classes}(\texttt{P})$.

Some members of $\textit{flow-information}(\texttt{P})$ are not valid approximations of the results of evaluating expressions in `P` and do not support our program transformation. The flow analyses with the desired properties are the ones that are both *acceptable* and *type respecting.* (The term “type respecting” was coined by Jagannathan et al. [28].) Intuitively, an acceptable analysis contains sets that are *big* enough, in that it correctly approximates the results of evaluating expressions. A type-respecting analysis contains sets that are *small* enough, in that it is at least as precise as the static type system, that is, each flow only contains classes that are subclasses of the corresponding static type. For a program `P`, we define:

$$\begin{aligned}\textit{acceptable}(\texttt{P}) = \{\ \varphi \in \textit{flow-information}(\texttt{P})\ \mid \\ \varphi \text{ satisfies the conditions listed in Figure 4 }\}\end{aligned}$$

$$\begin{aligned}
\mathit{type\text{-}respecting}(\mathtt{P}) = \{\ & \varphi \in \mathit{flow\text{-}information}(\mathtt{P}) \ \mid \\
& \forall \ell \in \mathit{labels}(\mathtt{P}) : \\
& \varphi(\ell) \subseteq \mathit{subclasses}(\mathtt{P}, \mathit{static\text{-}type}(\ell, \mathtt{P}))\ \} \\
\mathit{flow\text{-}analysis}(\mathtt{P}) = {} & \mathit{acceptable}(\mathtt{P}) \cap \mathit{type\text{-}respecting}(\mathtt{P}).
\end{aligned}$$

The conditions in Figure 4 for a flow analysis to be acceptable are somewhat unusual. The design of those conditions is influenced by the way the program transformation will use flow information to change type annotations: for a program point with label ℓ, the transformation uses the least upper bound of $\varphi(\ell)$, written $\sqcup\varphi(\ell)$, as the new type annotation. With that in mind, here is a closer look at the rules in Figure 4.

Rules (26)–(37) are related to one way of specifying 0-CFA [15,22,19]. The unusual aspect of them is that they are a mixture of subset constraints [22] and equality constraints [19]. If the sole purpose were to approximate the results of evaluating expressions, then all of the equality constraints can be relaxed to be subset constraints; the result would be 0-CFA. The reason for using equality constraints in some cases is to align the flow analysis with the type system. The type system does not have a general subsumption rule that allows subtyping to be used everywhere. Rather, in the type rules in Figure 3, subtyping is used in four places: Rule (19) for `new`-expressions, Rule (21) for calls, Rule (22) for static calls, and Rule (23) for method typing. In each case, there is a subset constraint in the corresponding rule for acceptable flow analyses in Figure 4: Rule (27) for `new`-expressions, Rule (30) for dynamic method calls, Rule (33) for static method calls, and Rule (37) for method typing. In contrast, Rule (18) for field selection requires the type of the field to *equal* the type of the field-selection expression; this is matched by the equality constraint in Rule (26). A similar comment applies to Rules (31) and (34). If there were a general subsumption rule, then that would allow subtyping to be used in the three mentioned cases where it is not allowed in the current definition of Java.

Rules (29), (32), and (35) have no direct counterparts in the type system and are needed to ensure that the flow analysis approximates the results of evaluating expressions. Specifically, Rule (29) models that a type cast to `C` only can be an object of `C` or a subclass of `C`, while Rule (32) and Rule (35) model that a receiver expression will become the `this`-object in the body of the called method. In Rule (32), the intersection with $\mathit{subclasses}(\mathtt{P}, \mathtt{D})$ ensures that the flow set for the `this` variable will only contain elements of $\mathit{subclasses}(\mathtt{P}, \mathtt{D})$. This intersection is correct, because objects of other classes will have a different implementation of `m` than `D::m`, so will not flow to ℓ''; it is needed because Rule (23) requires `this` to have type `D`. In contrast, such an intersection is not needed in Rule (35) because Rule (22) guarantees that the receiver expression has a type that is a subtype of the class that defines the called method.

Rule (36) is rather conservative: it says that the `this` object always can be an object of the class in which `this` occurs. The rule is needed because of Rule (23) for method typing, which asserts that `this` has type `C`. Finally, Rules (38) and (39) ensure that the signature of a method and the signature of an overriding method are the same. First, notice that Rule (40) ensures that least upper bounds are of a nonempty set. This rule is not needed only to ensure that least upper bounds exist, and is not a rule in most flow analyses.

A variant of Class Hierarchy Analysis [12] (CHA) can be defined as follows:

$$\mathit{CHA}(\mathtt{P}) = \lambda\ell.\mathit{subclasses}(\mathtt{P}, \mathit{static\text{-}type}(\ell, \mathtt{P})).$$

It is straightforward to show that $\mathit{CHA}(\mathtt{P})$ is the coarsest flow analysis of `P`, as stated in the following theorem.

Theorem 2 (CHA) *$\mathit{CHA}(\mathtt{P})$ is the greatest element of flow-analysis$(\mathtt{P})$.*

Note that *flow-analysis*$(\mathtt{P})$ does not have a least element. This is due to Rule (40) that requires all flows to be nonempty. Without it, *flow-analysis*$(\mathtt{P})$ always has a least element. This is because constraints of the forms used in Rules (26)–(39) always have a least solution [15]. The least solution can be found in $O(n^3)$ time where n is the size of the program from which the constraints were generated.

If Java had a bottom type, then this type could be used as the least upper bound of the empty set and Rule (40) would not be needed. Furthermore, *flow-analysis*$(\mathtt{P})$ would be a meet semilattice with both a greatest and least element. However, Java does not have a bottom type, so we have kept this constraint.

The property of being a flow analysis is preserved during computation, as stated in the following theorem, which is proved in Appendix A. (Palsberg [22] proved a similar result for the λ-calculus.)

Theorem 3 (Flow Preservation) *If $\varphi \in$ flow-analysis$(\mathtt{P}_1)$ and $\mathtt{P}_1 \mapsto \mathtt{P}_2$, then $\varphi \in$ flow-analysis$(\mathtt{P}_2)$ (technically φ restricted to the labels of $\mathtt{P}_2$).*

It is straightforward to compute $\mathit{CHA}(\mathtt{P})$. However, since $\mathit{CHA}(\mathtt{P})$ is the greatest element of *flow-analysis*$(\mathtt{P})$, it is the most conservative choice of flow analysis and will lead to the least number of inlinings. This raises the question of whether other polynomial-time algorithms could do better. The main difficulty is that *flow-analysis*$(\mathtt{P})$ does not have a least element, so there is not a unique best choice of flow analysis that improves on $\mathit{CHA}(\mathtt{P})$.

To illustrate that indeed there is a better polynomial time algorithm, we now define a flow analysis with mixed constraints and nonempty sets; the analysis

- for each $\texttt{e.f}^\ell$ in $\texttt{P}$: $\quad \varphi(\mathit{lab}(\texttt{f})) = \varphi(\ell) \qquad (26)$
- for each $\texttt{new}^\ell\ \texttt{C}(\overline{\texttt{e}})$ in $\texttt{P}$, where $\mathit{fields}(\overline{\texttt{CD}}, \texttt{C}) = \overline{\texttt{D}}\ \overline{\texttt{f}}$:

$$\varphi(\mathit{lab}(\overline{\texttt{e}})) \subseteq \varphi(\mathit{lab}(\overline{\texttt{f}})) \qquad (27)$$
$$\texttt{C} \in \varphi(\ell) \qquad (28)$$

- for each $(\texttt{C})^\ell\texttt{e}$ in $\texttt{P}$:

$$\varphi(\mathit{lab}(\texttt{e})) \cap \mathit{subclasses}(\texttt{P}, \texttt{C}) \subseteq \varphi(\ell) \qquad (29)$$

- for each $\texttt{e.m}(\overline{\texttt{d}})^\ell$ in $\texttt{P}$ and each class $\texttt{C}$ in $\texttt{P}$, where $\mathit{mbody}(\overline{\texttt{CD}}, \texttt{C}, \texttt{m}) = (\ell', \overline{\texttt{x}}, \texttt{e}')$, $\mathit{impl}(\overline{\texttt{CD}}, \texttt{C}, \texttt{m}) = \texttt{D::m}$, and ℓ'' is the label for $\texttt{D}$'s $\texttt{this}$ occurrences:

$$\texttt{C} \in \varphi(\mathit{lab}(\texttt{e})) \Rightarrow \varphi(\mathit{lab}(\overline{\texttt{d}})) \subseteq \varphi(\mathit{lab}(\overline{\texttt{x}})) \qquad (30)$$
$$\texttt{C} \in \varphi(\mathit{lab}(\texttt{e})) \Rightarrow \varphi(\ell') = \varphi(\ell) \qquad (31)$$
$$\texttt{C} \in \varphi(\mathit{lab}(\texttt{e})) \Rightarrow \varphi(\mathit{lab}(\texttt{e})) \cap \mathit{subclasses}(\texttt{P}, \texttt{D}) \subseteq \varphi(\ell'') \qquad (32)$$

- for each $\texttt{e.C::m}(\overline{\texttt{d}})^\ell$ in $\texttt{P}$, where $\mathit{mbody}(\overline{\texttt{CD}}, \texttt{C}, \texttt{m}) = (\ell', \overline{\texttt{x}}, \texttt{e}')$, $\mathit{impl}(\overline{\texttt{CD}}, \texttt{C}, \texttt{m}) = \texttt{D::m}$, and ℓ'' is the label for $\texttt{D}$'s $\texttt{this}$ occurrences

$$\varphi(\mathit{lab}(\overline{\texttt{d}})) \subseteq \varphi(\mathit{lab}(\overline{\texttt{x}})) \qquad (33)$$
$$\varphi(\ell') = \varphi(\ell) \qquad (34)$$
$$\varphi(\mathit{lab}(\texttt{e})) \subseteq \varphi(\ell'') \qquad (35)$$

- for each class $\texttt{C}$ in $\texttt{P}$, where ℓ is the label for $\texttt{C}$'s $\texttt{this}$ occurrences:

$$\texttt{C} \in \varphi(\ell) \qquad (36)$$

- for each method in $\texttt{P}$, with body $\{\texttt{return}^\ell\texttt{e}_0;\}$

$$\varphi(\mathit{lab}(\texttt{e}_0)) \subseteq \varphi(\ell) \qquad (37)$$

- for each method name $\texttt{m}$ declared or inherited in class $\texttt{C}$ of $\texttt{P}$, if

$$\overline{\texttt{CD}}(\texttt{C}) = \texttt{class C extends C}_0\ \{\overline{\texttt{D}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\}$$
$$\mathit{mbody}(\overline{\texttt{CD}}, \texttt{C}, \texttt{m}) = (\ell_1, \overline{\texttt{x}}_1, \texttt{e}_1)$$
$$\mathit{mbody}(\overline{\texttt{CD}}, \texttt{C}_0, \texttt{m}) = (\ell_2, \overline{\texttt{x}}_2, \texttt{e}_2)$$

then

$$\varphi(\mathit{lab}(\overline{\texttt{x}}_1)) = \varphi(\mathit{lab}(\overline{\texttt{x}}_2)) \qquad (38)$$
$$\varphi(\ell_1) = \varphi(\ell_2) \qquad (39)$$

- for each $\ell \in \mathit{labels}(\texttt{P})$: $\quad \varphi(\ell) \neq \emptyset \qquad (40)$

Fig. 4. Requirements for an acceptable flow analyses φ of a program $\texttt{P} = (\overline{\texttt{CD}}, \texttt{e})$.

is called $MN(\texttt{P})$ (for Mixed and Nonempty). First, the notion of *lifting* a flow analysis is:

$$\begin{aligned}
&\mathit{lift}(\varphi) \,:\, \mathit{flow\text{-}information}(\texttt{P}) \rightarrow \mathit{flow\text{-}information}(\texttt{P}) \\
&\mathit{lift}(\varphi) = \lambda\ell. \begin{cases} \varphi(\ell) & \text{if } \sqcup\varphi(\ell) \text{ exists} \\ \{\ \mathit{static\text{-}type}(\ell,\texttt{P})\ \} & \text{otherwise} \end{cases}
\end{aligned}$$

Notice that $\mathit{lift}(\varphi)(\ell) \neq \emptyset$ for all $\ell \in \mathit{labels}(\texttt{P})$. For Featherweight Java and FJS, $\sqcup\varphi(\ell)$ exists for all nonempty sets $\varphi(\ell)$. The full Java type system enables multiple subtyping among interfaces which can lead to nonempty flows without a least upper bound.

Second, the definition of $MN(\texttt{P})$ is:

$$\begin{aligned}
\varphi_0 &= \text{the least flow analysis satisfying Rules (26)–(39)} \\
MN(\texttt{P}) &= \text{the least flow analysis greater than } \mathit{lift}(\varphi_0) \\
&\ \text{satisfying Rules (26)–(39)}
\end{aligned}$$

This algorithm is polynomial time: φ_0 takes polynomial time to compute using the technique of Fähndrich and Aiken [23], which intuitively is a fixed-point computation with $\lambda\ell.\emptyset$ as the starting point. Lifting clearly takes polynomial time, and $MN(\texttt{P})$ takes polynomial time to compute by using the Fähndrich-Aiken algorithm again, but this time with $\mathit{lift}(\varphi_0)$ as the starting point for the fixed-point computation. This two-step procedure makes $\sqcup MN(\texttt{P})(\ell)$ equal to $\mathit{static\text{-}type}(\ell,\texttt{P})$ for any $\ell \in \mathit{labels}(\texttt{P})$ such that $\varphi_0(\ell) = \emptyset$. Thus the program transformation based on $MN(\texttt{P})$ will not change the type annotation for the program points labeled by such ℓ.

It is left to future work to settle whether there is a way to avoid both Rule (40) and the two-step procedure of $MN()$, by somehow mapping the empty flow set to a legal Java type.

There might be worthwhile elements of *flow-analysis*($\texttt{P}$) other than $CHA(\texttt{P})$ and $MN(\texttt{P})$. Any element of *flow-analysis*($\texttt{P}$) can be used as an argument to the program transformation, which we present next.

4 Program Transformation

The program transformation is parameterised by a flow analysis, and it operates on program fragments and type environments in a compositional fashion. It transforms each program fragment into a similar program fragment with the same label, and it transforms each type environment into a type environment which defines the same variables. The changes made are that:

- it changes some dynamic method calls to static method calls,
- it changes the type annotations, and
- it changes the classes used in type casts.

In each case, the change is made on the basis of the supplied flow analysis. Specifically, (1) a dynamic call is changed to a static call when the flow analysis determines that there is a unique target method and (2) a type annotation and the class in a type cast are changed to the least upper bound of the classes in the corresponding flow. Taking the least upper bound of the classes in a flow is justified as follows. The transformation is restricted to flow analyses that are acceptable and type respecting, which means the following. First, all flows are nonempty. Second, nonempty sets of classes admit least upper bounds because FJS is a single-inheritance language. Third, the type-respecting property implies that the new types (that is, the least upper bounds) can only be more refined than the old ones.

The transformation consists of the following cases:

Transformation	Meaning
$[\![\mathtt{P}]\!]_{\varphi}$	the transformation of $\mathtt{P}$ using φ
$[\![\mathtt{CD}]\!]_{\varphi}^{\overline{\mathtt{CD}}}$	the transformation of $\mathtt{CD}$ using φ and $\overline{\mathtt{CD}}$
$[\![\mathtt{K}]\!]_{\varphi}$	the transformation of $\mathtt{K}$ using φ
$[\![\mathtt{M}]\!]_{\varphi}^{\overline{\mathtt{CD}}}$	the transformation of $\mathtt{M}$ using φ and $\overline{\mathtt{CD}}$
$[\![\mathtt{e}]\!]_{\varphi}^{\overline{\mathtt{CD}}}$	the transformation of $\mathtt{e}$ using φ and $\overline{\mathtt{CD}}$
$[\![\Gamma]\!]_{\varphi}$	the transformation of Γ using φ

The definition of the transformation appears in Figure 5.

We now present four correctness theorems: the transformation preserves typability, the transformation is operationally correct, a flow analysis of the original program is also a flow analysis of the transformed program, and the transformation is idempotent. First our main result, which is proved in Appendix B.

Theorem 4 (Typability Preservation) *Suppose* $\varphi \in \mathit{flow\text{-}analysis}(\mathtt{P})$ *and* $\mathtt{P} = (\overline{\mathtt{CD}},\mathtt{e})$. *If* $\vdash \mathtt{P} \in \mathtt{C}$ *then* $\vdash [\![\mathtt{P}]\!]_{\varphi} \in \sqcup\varphi(\mathit{lab}(\mathtt{e}))$.

The transformation is also operationally correct, in that the transformed program simulates the original program step for step and vice versa, as stated in

$$
\begin{aligned}
&[\![\texttt{(}\overline{\texttt{CD}}\texttt{,e)}]\!]_{\varphi} &&= \texttt{(}[\![\overline{\texttt{CD}}]\!]_{\varphi}^{\overline{\texttt{CD}}}\texttt{,}[\![\texttt{e}]\!]_{\varphi}^{\overline{\texttt{CD}}}\texttt{)}\\
&[\![\texttt{class C extends C}_0\ \texttt{\{}\overline{\texttt{D}}\ \overline{\texttt{f}^{\ell}}\texttt{; K }\overline{\texttt{M}}\texttt{\}}]\!]_{\varphi}^{\overline{\texttt{CD}}} &&= \texttt{class C extends C}_0\\
& &&\quad \texttt{\{}\sqcup\varphi(\overline{\ell})\ \overline{\texttt{f}^{\ell}}\texttt{; }[\![\texttt{K}]\!]_{\varphi}\ [\![\overline{\texttt{M}}]\!]_{\varphi}^{\overline{\texttt{CD}}}\texttt{\}}\\
&[\![\texttt{C(}\overline{\texttt{D}}\ \overline{\texttt{f}}\texttt{) \{super(}\overline{\texttt{f1}}\texttt{); this.}\overline{\texttt{f2=f2}}\texttt{\}}]\!]_{\varphi} &&= \texttt{C(}\sqcup\varphi(\mathit{lab}(\overline{\texttt{f}}))\ \overline{\texttt{f}}\texttt{)}\\
& &&\quad \texttt{\{super(}\overline{\texttt{f1}}\texttt{); this.}\overline{\texttt{f2=f2}}\texttt{\}}\\
&[\![\texttt{D m(}\overline{\texttt{E}}\ \overline{\texttt{x}^{\ell}}\texttt{) \{return}^{\ell}\texttt{e;\}}]\!]_{\varphi}^{\overline{\texttt{CD}}} &&= \sqcup\varphi(\ell)\ \texttt{m(}\sqcup\varphi(\overline{\ell})\ \overline{\texttt{x}^{\ell}}\texttt{)}\\
& &&\quad \texttt{\{return}^{\ell}[\![\texttt{e}]\!]_{\varphi}^{\overline{\texttt{CD}}}\texttt{;\}}
\end{aligned}
$$

$$
\begin{aligned}
&[\![\texttt{x}^{\ell}]\!]_{\varphi}^{\overline{\texttt{CD}}} &&= \texttt{x}^{\ell}\\
&[\![\texttt{e.f}^{\ell}]\!]_{\varphi}^{\overline{\texttt{CD}}} &&= [\![\texttt{e}]\!]_{\varphi}^{\overline{\texttt{CD}}}\texttt{.f}^{\ell}\\
&[\![\texttt{new}^{\ell}\ \texttt{D(}\overline{\texttt{e}}\texttt{)}]\!]_{\varphi}^{\overline{\texttt{CD}}} &&= \texttt{new}^{\ell}\ \texttt{D(}[\![\overline{\texttt{e}}]\!]_{\varphi}^{\overline{\texttt{CD}}}\texttt{)}\\
&[\![\texttt{(D)}^{\ell}\texttt{e}]\!]_{\varphi}^{\overline{\texttt{CD}}} &&= \texttt{(}\sqcup\varphi(\ell)\texttt{)}^{\ell}[\![\texttt{e}]\!]_{\varphi}^{\overline{\texttt{CD}}}\\
&[\![\texttt{e.m(}\overline{\texttt{d}}\texttt{)}^{\ell}]\!]_{\varphi}^{\overline{\texttt{CD}}} &&= [\![\texttt{e}]\!]_{\varphi}^{\overline{\texttt{CD}}}\texttt{.D::m(}[\![\overline{\texttt{d}}]\!]_{\varphi}^{\overline{\texttt{CD}}}\texttt{)}^{\ell}\\
&\quad \text{where } \forall \texttt{E} \in \varphi(\mathit{lab}(\texttt{e})) : \mathit{impl}(\overline{\texttt{CD}},\texttt{E},\texttt{m}) = \texttt{D::m}\\
&[\![\texttt{e.m(}\overline{\texttt{d}}\texttt{)}^{\ell}]\!]_{\varphi}^{\overline{\texttt{CD}}} &&= [\![\texttt{e}]\!]_{\varphi}^{\overline{\texttt{CD}}}\texttt{.m(}[\![\overline{\texttt{d}}]\!]_{\varphi}^{\overline{\texttt{CD}}}\texttt{)}^{\ell}\\
&\quad \text{otherwise}\\
&[\![\texttt{e.D::m(}\overline{\texttt{d}}\texttt{)}^{\ell}]\!]_{\varphi}^{\overline{\texttt{CD}}} &&= [\![\texttt{e}]\!]_{\varphi}^{\overline{\texttt{CD}}}\texttt{.D::m(}[\![\overline{\texttt{d}}]\!]_{\varphi}^{\overline{\texttt{CD}}}\texttt{)}^{\ell}
\end{aligned}
$$

$$
\begin{aligned}
&[\![\texttt{x}_1 : \texttt{C}_1, \ldots, \texttt{x}_n : \texttt{C}_n]\!]_{\varphi} &&= \texttt{x}_1 : \sqcup\varphi(\mathit{lab}(\texttt{x}_1)), \ldots,\\
& &&\quad \texttt{x}_n : \sqcup\varphi(\mathit{lab}(\texttt{x}_n))
\end{aligned}
$$

Fig. 5. The Transformation of Dynamic to Static Dispatch

the following theorem, which is proved in Appendix C. Operational correctness for a multistep computation follows from Theorems 3 and 5.

Theorem 5 (Operational Correctness) *If* $\varphi \in \mathit{flow\text{-}analysis}(\texttt{P}_1)$ *then:* $\texttt{P}_1 \longmapsto \texttt{P}_2$ *if and only if* $[\![\texttt{P}_1]\!]_{\varphi} \longmapsto [\![\texttt{P}_2]\!]_{\varphi}$.

It is straightforward to prove that a flow analysis of a program is also a flow analysis of the transformed program, as stated in the following theorem.

Theorem 6 (Analysis Preservation) *If* $\varphi \in \mathit{flow\text{-}analysis}(\texttt{P})$*, then* $\varphi \in \mathit{flow\text{-}analysis}([\![\texttt{P}]\!]_{\varphi})$.

```
                  class A {
                     void m(Q arg) {         class Q {
A x = new A();          arg.p();                void p() {...}
B y = new B();       }                       }
                  }                          class S extends Q {
x.m(new Q());     class B extends A {           void p() {...}
y.m(new S());        void m(Q arg) {...}     }
                  }
```

Fig. 6. Example program

Given a flow analysis, it is sufficient to apply the transformation only once: applying the transformation again with the *same* flow analysis will not lead to any further change. We can state this as the following idempotence property of the transformation, which is straightforward to prove.

Theorem 7 (Idempotence) *If* $\varphi \in \mathit{flow\text{-}information}(\texttt{P})$ *then* $[\![[\![\texttt{P}]\!]_\varphi]\!]_\varphi = [\![\texttt{P}]\!]_\varphi$.

5 Examples

We first present an example that illustrates the difference between *CHA*(P), *0-CFA*(P), and our analysis *MN*(P). Consider the example program in Figure 6. There are four classes: A, B, Q, and S, where B extends A and where S extends Q. Notice that A has a method m, and that B also has a method m that overrides the one from A. Similarly Q has a method p, and S also has a method p that overrides the one from Q. Among these methods, only the body of A.m is of interest here, while the bodies of the other methods are left unspecified. To the left of the classes are four lines of code that should be seen as being part of some other class. The first two lines are declarations of fields x and y, and the last two lines are method calls.

Now consider which of the method calls in the program will be inlined based on *CHA*(P), *0-CFA*(P), and our analysis *MN*(P).

First consider *CHA*(P). In the first method call `x.m(new Q())`, the static type of x is A. Since A has a subclass B, $\mathit{CHA}(\texttt{P})(\mathit{lab}(\texttt{x})) = \{\texttt{A}, \texttt{B}\}$. Note that A and B have different implementations of m, so *CHA*(P) does not lead to inlining of the method call `x.m(new Q())`.

By the way, for the example program, Bacon and Sweeney's Rapid Type Analysis (RTA) [29,30] gives the same result as CHA. RTA is similar to CHA except that its flow sets contain only classes that are actually instantiated in the program [31]. In the example program, objects are created from all four classes, so there is no difference between RTA and CHA in this case.

```
A x = new A();
Q y;
if (...) {
   y = new S();
} else {
   y = x.m();
}
y.p();
```

```
class A {
   Q m() {
     // infinite
     // recursion!
     return this.m();
   }
}
```

```
class Q {
   void p() {...}
}
class S extends Q {
   void p() {...}
}
```

Fig. 7. Example that shows empty flow sets

In the second method call `y.m(new S())`, the static type of `y` is `B`. Since `B` has no subclasses, $\mathit{CHA}(\texttt{P})(\mathit{lab}(\texttt{y})) = \{\texttt{B}\}$. Clearly, `B` has just one implementation of `m`, so $\mathit{CHA}(\texttt{P})$ leads to inlining of the method call `x.m(new Q())`.

In the third method call `arg.p()`, the static type of `arg` is `Q`. Since `Q` has a subclass `S`, $\mathit{CHA}(\texttt{P})(\mathit{lab}(\texttt{arg})) = \{\texttt{Q}, \texttt{S}\}$. Note that `Q` and `S` have different implementations of `p`, so $\mathit{CHA}(\texttt{P})$ will not lead to inlining of the method call `arg.p()`.

Second consider $\mathit{0\text{-}CFA}(\texttt{P})$. The initialisations of `x` and `y` show that an `A`-object flows to `x` and that a `B`-object flows to `y`. So, in the first method call `x.m(new Q())`, $\mathit{0\text{-}CFA}(\texttt{P})(\mathit{lab}(\texttt{x})) = \{A\}$, and in the second method call `y.m(new S())`, $\mathit{0\text{-}CFA}(\texttt{P})(\mathit{lab}(\texttt{y})) = \{B\}$, and thus $\mathit{0\text{-}CFA}(\texttt{P})$ leads to inlining of both those method calls. In the third method call `arg.p()`, the receiver is `arg`, and the only value that flows to `arg` is the `Q`-object from the call site `x.m(new Q())`. So, $\mathit{0\text{-}CFA}(\texttt{P})$ also leads to inlining of the third method call.

Third consider our analysis $\mathit{MN}(\texttt{P})$, which, by construction, is at least as precise as $\mathit{CHA}(\texttt{P})$ and at most as precise as $\mathit{0\text{-}CFA}(\texttt{P})$. For the first two method calls, it is straightforward to see that $\mathit{MN}(\texttt{P})$ gives the same results as $\mathit{0\text{-}CFA}(\texttt{P})$ for `x` and `y`, Hence, $\mathit{MN}(\texttt{P})$ leads to inlining of both those method calls.

However, $\mathit{0\text{-}CFA}(\texttt{P})$ and $\mathit{MN}(\texttt{P})$ differ on the third method call `arg.p()`. Here, Rule (38) forces the flow sets for `arg` in `A.m` to be the same as for `arg` in `B.m`. The second call site `y.m(new S())` shows that an `S`-object flows to `arg` in `B.m`. So, the unified flow set for both `arg` in `A.m` and `arg` in `B.m` is $\{\texttt{Q}, \texttt{S}\}$, and hence the call site is not inlined.

In conclusion, $\mathit{CHA}(\texttt{P})$ leads to the inlining of one call site, $\mathit{MN}(\texttt{P})$ leads to the inlining of two call sites, while $\mathit{0\text{-}CFA}(\texttt{P})$ leads to the inlining of three call sites.

As a final example, consider the program in Figure 7.

In this example, class `S` extends `Q` and both have a `p` method. Notice that variable `y` is initialised either with an instance of `S` or with the result of `new A().m()`, which never returns. Clearly in any run of this program only an instance of `S` will reach the call site `y.p()`, so it is safe to inline `S::p` at this point. Since `A::m` does not return, $\mathit{0\text{-}CFA}(\mathtt{P})(\mathit{lab}(\mathtt{A::m})) = \emptyset$, so in turn, $\mathit{0\text{-}CFA}(\mathtt{P})(\mathit{lab}(\mathtt{y})) = \{\mathtt{S}\}$. Thus $\mathit{0\text{-}CFA}(\mathtt{P})$ leads to inlining of the call site. However, our analysis will return $\mathit{MN}(\mathtt{P})(\mathit{lab}(\mathtt{A::m})) = \{\mathtt{Q}, \mathtt{S}\}$, so in turn, $\mathit{MN}(\mathtt{P})(\mathit{lab}(\mathtt{y})) = \{\mathtt{Q}, \mathtt{S}\}$. Thus $\mathit{MN}(\mathtt{P})$ does not lead to inlining of the call site.

Observe that, in both examples, our analysis did not inline some call sites because of constraints imposed by the type system. In the first example, the constraints were due to Java's invariant subtyping for method parameters and returns. In the second example, the constraints were due to Java's lack of a bottom type. Based on the result of Palsberg and O'Keefe [17], one can imagine a more expressive type system for Java that would admit $\mathit{0\text{-}CFA}(\mathtt{P})$ as a type-preserving analysis, and that would allow all of the inlining discussed above. In particular, such a type system would likely have a bottom type and likely have depth subtyping for methods.

6 Conclusion

We have shown how to inline methods while preserving typability in a single-inheritance language without resorting to the insertion of type casts or new types. Our approach is based on flow analysis, and we found it tricky to get the requirements for the flow analysis right. During the process of proving correctness, we discovered the need for flow constraints that would not usually be used in a flow analysis, e.g., Rule (36). The requirement that all flow sets must be nonempty is unusual, and it entails that there is no unique best analysis that satisfies the requirements. While CHA and our own *MN* analysis satisfy the requirements, more work is needed to investigate alternatives.

Concerning the effectiveness of the various analyses, our analysis *MN* is always at least as precise as *CHA* and at most as precise as *0-CFA*. This is because *MN* imposes fewer flow constraints than *CHA* and more flow constraints than *0-CFA*.

One can evaluate the practical effectiveness of an inlining strategy in at least two ways: the static count of inlined calls and the run-time count of inlined calls. With regards to the static count of inlined calls, *CHA* is an excellent baseline. Tip and Palsberg [31] showed that *RTA* (the variant of *CHA* discussed earlier) inlines 92.2% of all virtual call sites in a large suite of Java programs. Thus, even though one can try better analyses, the room for improvement is just the remaining 7.8% of the virtual call sites. Tip and Palsberg did experiment with other analyses than *RTA*, but even their most powerful analysis inlined just 93.0% of all virtual call sites. Thus, even though *MN* is better than *CHA* and *RTA*, we conjecture that there will be a small difference in the number of call sites that will be inlined. The important property of *MN* is that it guarantees that inlining can be done without the insertion of type casts.

In practice, the run-time count of inlined calls is more important than the static count. For example, the inlining of a method call in an inner loop can have a major impact on the run-time performance. So, even though an analysis might inline less than one percent more of the statically-counted call sites, some of those calls might occur in frequently executed code and therefore be important to inline. Sundaresan et al. [32] investigated an analysis called *VTA* which is more powerful that *CHA*, and they found that their analysis leads to nontrivial improvements in run-time performance compared to *CHA*. They conclude: "some of the extra calls sites found by *VTA* could be important ones for inlining."

One idea for future work is to do *two* flow analyses of a program: one which is typability preserving, and one that is more powerful but not necessarily typability preserving. The difference between the two sets of call sites suggested for inlining can then be inlined *with* type casts, in the style of Wright et al. [7]. In this way, the performance penalty of the type casts is only payed when deemed necessary.

Another idea, due to Ralf Laemmel, is to change the type system such that 0-CFA would lead to a type preserving transformation, perhaps with inspiration from the equivalence result of Palsberg and O'Keefe [17].

Acknowledgements. We thank the 84 students who took Palsberg's graduate course on programming languages in 1999-2001 and tried, as a homework, to find and implement a solution to the problem of type-safe method inlining for a subset of Java. Only 10 of the implementations seemed not to have errors, leading to the realisation that the problem is considerably harder than flow-directed inlining for an untyped language.

A preliminary version of this paper was presented at ECOOP'02, European Conference on Object-Oriented Programming 2002. We thank Ralf Laemmel, Mayur Naik, and the anonymous referees for helpful comments on a draft of the paper. We also thank Ben Titzer for finding a mistake in an earlier version of the flow analysis.

Palsberg is supported by a National Science Foundation Faculty Early Career Development Award, CCR–9734265.

A Proof of Theorem 3

First, observe that $\mathit{labels}(\mathtt{P}_2) \subseteq \mathit{labels}(\mathtt{P}_1)$ so φ (restricted to $\mathit{labels}(\mathtt{P}_2)$) is a flow analysis of $\mathtt{P}_2$. Let $\mathtt{P}_i = (\overline{\mathtt{CD}}, \mathtt{X}\langle \mathtt{e}_i \rangle)$ where $\mathtt{e}_1$ and $\mathtt{e}_2$ are as in the rules of Figure 2. Since $\mathtt{P}_1$ and $\mathtt{P}_2$ differ only in $\mathtt{e}_1$ and $\mathtt{e}_2$, φ satisfies the conditions for acceptability and type respecting for $\mathtt{P}_2$ except for the conditions on $\mathtt{e}_2$ and its subexpressions that are not subexpressions of $\mathtt{e}_1$, and on the expression that $\mathtt{e}_1$ appears immediately within. By inspection of the rules, the last condition will hold if $\varphi(\mathit{lab}(\mathtt{e}_2)) \subseteq \varphi(\mathit{lab}(\mathtt{e}_1))$. Thus, we need just to show the latter and that φ satisfies the conditions for $\mathtt{e}_2$ and its subexpressions that are not subexpressions of $\mathtt{e}_1$. Consider the various cases for the reduction rule.

field selection: In this case $\mathtt{e}_1 = \mathtt{new}^{\ell_1}\ \mathtt{C}(\overline{\mathtt{e}}).\mathtt{f}_i^{\ell_2}$, $\mathtt{e}_2 = \mathtt{e}_i$, and $\mathit{fields}(\overline{\mathtt{CD}}, \mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}}$. Thus, $\mathtt{e}_2$ is a subexpression of $\mathtt{e}_1$. By Rule 27, $\varphi(\mathit{lab}(\mathtt{e}_i)) \subseteq \varphi(\mathit{lab}(\mathtt{f}_i))$; by Rule 26, $\varphi(\mathit{lab}(\mathtt{f}_i)) = \varphi(\ell_2)$. By transitivity, $\varphi(\mathit{lab}(\mathtt{e}_2)) = \varphi(\mathit{lab}(\mathtt{e}_i)) \subseteq \varphi(\ell_2) = \varphi(\mathit{lab}(\mathtt{e}_1))$, as required.

cast: In this case $\mathtt{e}_1 = (\mathtt{D})^{\ell_1}\ \mathtt{new}^{\ell_2}\ \mathtt{C}(\overline{\mathtt{e}})$, $\mathtt{e}_2 = \mathtt{new}^{\ell_2}\ \mathtt{C}(\overline{\mathtt{e}})$, and $\overline{\mathtt{CD}} \vdash \mathtt{C} <: \mathtt{D}$. Thus, $\mathtt{e}_2$ is a subexpression of $\mathtt{e}_1$. By the type respecting property and Rule 19, $\varphi(\ell_2) \subseteq \mathit{subclasses}(\mathtt{P}_1, \mathtt{C})$. Since $\overline{\mathtt{CD}} \vdash \mathtt{C} <: \mathtt{D}$, $\mathit{subclasses}(\mathtt{P}_1, \mathtt{C}) \subseteq \mathit{subclasses}(\mathtt{P}_2, \mathtt{D})$, so:

$$\varphi(\ell_2) \cap \mathit{subclasses}(\mathtt{P}_1, \mathtt{D}) = \varphi(\ell_2) \tag{A.1}$$

By Rule 29, $\varphi(\ell_2) \cap \mathit{subclasses}(\mathtt{P}_1, \mathtt{D}) \subseteq \varphi(\ell_1)$. Thus by (A.1), $\varphi(\ell_2) \subseteq \varphi(\ell_1)$, as required.

dynamic method call: In this case both $\mathtt{e}_1 = \mathtt{new}^{\ell_1}\ \mathtt{C}(\overline{\mathtt{e}}).\mathtt{m}(\overline{\mathtt{d}})^{\ell_2}$ and $\mathtt{e}_2 = \mathtt{e}_0\{\mathtt{this}, \overline{\mathtt{x}} := \mathtt{new}^{\ell_1}\ \mathtt{C}(\overline{\mathtt{e}}), \overline{\mathtt{d}}\}$ where $\mathit{mbody}(\overline{\mathtt{CD}}, \mathtt{C}, \mathtt{m}) = (\ell', \overline{\mathtt{x}}, \mathtt{e}_0)$. By Rule 28, $\mathtt{C} \in \varphi(\ell_2)$. So by Rule 30, $\varphi(\mathit{lab}(\overline{\mathtt{d}})) \subseteq \varphi(\mathit{lab}(\overline{\mathtt{x}}))$, and by Rule 32,

$$\varphi(\ell_1) \cap \mathit{subclasses}(\mathtt{D}, \mathtt{P}) \subseteq \varphi(\ell'')$$

where $\mathit{impl}(\overline{\mathtt{CD}}, \mathtt{C}, \mathtt{m}) = \mathtt{D}::\mathtt{m}$ and ℓ'' is the label for $\mathtt{D}$'s $\mathtt{this}$ occurances. Since φ satisfies the conditions for type respecting, and the static type of ℓ_1 must be $\mathtt{C}$ by Rule 19, $\varphi(\ell_1) \subseteq \mathit{subclasses}(\mathtt{C}, \mathtt{P})$. By the rules, $\overline{\mathtt{CD}} \vdash \mathtt{C} <: \mathtt{D}$, so $\mathit{subclasses}(\mathtt{C}, \mathtt{P}) \subseteq \mathit{subclasses}(\mathtt{D}, \mathtt{P})$. Thus $\varphi(\ell_1) \cap \mathit{subclasses}(\mathtt{D}, \mathtt{P}) = \varphi(\ell_1)$ and $\varphi(\ell_1) \subseteq \varphi(\ell'')$. By Lemma 8 (below), φ satisfies the conditions for acceptability and type respecting for $\mathtt{e}_2$ and all its subexpressions. By Rule 31, $\varphi(\ell') = \varphi(\ell_2)$; by Rule 37, $\varphi(\mathit{lab}(\mathtt{e}_0)) \subseteq \varphi(\ell')$. Also by Lemma 8, $\varphi(\mathit{lab}(\mathtt{e}_2)) \subseteq \varphi(\mathit{lab}(\mathtt{e}_0))$. Thus $\varphi(\mathit{lab}(\mathtt{e}_2)) \subseteq \varphi(\mathit{lab}(\mathtt{e}_0)) \subseteq \varphi(\ell') = \varphi(\ell_2) = \varphi(\mathit{lab}(\mathtt{e}_1))$, as required.

static method call: In this case both $\mathtt{e}_1 = \mathtt{new}^{\ell_1}\ \mathtt{C}(\overline{\mathtt{e}}).\mathtt{D}::\mathtt{m}(\overline{\mathtt{d}})^{\ell_2}$ and $\mathtt{e}_2 = \mathtt{e}_0\{\mathtt{this}, \overline{\mathtt{x}} := \mathtt{new}^{\ell_1}\ \mathtt{C}(\overline{\mathtt{e}}), \overline{\mathtt{d}}\}$ where $\mathit{mbody}(\overline{\mathtt{CD}}, \mathtt{D}, \mathtt{m}) = (\ell', \overline{\mathtt{x}}, \mathtt{e})$. By Rule 33, $\varphi(\mathit{lab}(\overline{\mathtt{d}})) \subseteq \varphi(\mathit{lab}(\overline{\mathtt{x}}))$, and by Rule 35, $\varphi(\ell_1) \subseteq \varphi(\mathit{lab}(\mathtt{this}))$. By Lemma 8 (below), φ satisfies the conditions for acceptability and type respecting for $\mathtt{e}_2$ and all its subexpressions. By Rule 34, $\varphi(\ell') = \varphi(\ell_2)$, and by Rule 37,

$\varphi(lab(\mathtt{e}_0)) \subseteq \varphi(\ell')$. Also by Lemma 8, $\varphi(lab(\mathtt{e}_2)) \subseteq \varphi(lab(\mathtt{e}_0))$. Thus

$$\varphi(lab(\mathtt{e}_2)) \subseteq \varphi(lab(\mathtt{e}_0)) \subseteq \varphi(\ell') = \varphi(\ell_2) = \varphi(lab(\mathtt{e}_1))$$

as required.

Lemma 8 *If φ satisfies the conditions for acceptability and type respecting for all labels in $\mathtt{e}$ and $\overline{\mathtt{d}}$ and if $\varphi(lab(\overline{\mathtt{d}})) \subseteq \varphi(lab(\overline{\mathtt{x}}))$ then φ satisfies the conditions for acceptability and type respecting for all labels in $\mathtt{e}\{\overline{\mathtt{x}} := \overline{\mathtt{d}}\}$ and $\varphi(lab(\mathtt{e}\{\overline{\mathtt{x}} := \overline{\mathtt{d}}\})) \subseteq \varphi(lab(\mathtt{e}))$.*

Proof. Straightforward.

B Proof of Theorem 4

Theorem 4 follows immediately from Rule (25), and from Lemma 9 and Lemma 11, as stated and proved below.

Lemma 9 *Suppose $\varphi \in acceptable(\mathtt{P}) \cap type\text{-}respecting(\mathtt{P})$, and $\mathtt{P} = (\overline{\mathtt{CD}}, \mathtt{e}_0)$. If $\overline{\mathtt{CD}} \vdash \overline{\mathtt{CD}}$ OK, then $[\![\overline{\mathtt{CD}}]\!]_{\varphi}^{\overline{\mathtt{CD}}} \vdash [\![\overline{\mathtt{CD}}]\!]_{\varphi}^{\overline{\mathtt{CD}}}$ OK.*

Proof. Immediate from Lemmas 10 and 14 (below), using Rule (24).

Lemma 10 *Suppose $\varphi \in acceptable(\mathtt{P}) \cap type\text{-}respecting(\mathtt{P})$, and $\mathtt{P} = (\overline{\mathtt{CD}}, \mathtt{e}_0)$. If $\overline{\mathtt{CD}} \vdash \overline{\mathtt{M}}$ OK in $\mathtt{C}$, then $[\![\overline{\mathtt{CD}}]\!]_{\varphi}^{\overline{\mathtt{CD}}} \vdash [\![\overline{\mathtt{M}}]\!]_{\varphi}^{\overline{\mathtt{CD}}}$ OK in $\mathtt{C}$.*

Proof. Straightforward from Lemmas 11, 12, 15, and 16 (below), using Rules (17), (23), (36), and (37).

Lemma 11 *Suppose $\varphi \in acceptable(\mathtt{P}) \cap type\text{-}respecting(\mathtt{P})$, and $\mathtt{P} = (\overline{\mathtt{CD}}, \mathtt{e}_0)$. If $\overline{\mathtt{CD}}; \Gamma \vdash \mathtt{e} \in \mathtt{D}$, then $[\![\overline{\mathtt{CD}}]\!]_{\varphi}^{\overline{\mathtt{CD}}}; [\![\Gamma]\!]_{\varphi} \vdash [\![\mathtt{e}]\!]_{\varphi}^{\overline{\mathtt{CD}}} \in \sqcup\varphi(lab(\mathtt{e}))$.*

Proof. We proceed by induction on the structure of the derivation of $\overline{\mathtt{CD}}; \Gamma \vdash \mathtt{e} \in \mathtt{D}$. There are six cases, depending on which one of Rules (17)–(22) was the last one used to derive $\overline{\mathtt{CD}}; \Gamma \vdash \mathtt{e} \in \mathtt{D}$.

- (17) $\mathtt{e} \equiv \mathtt{x}^{\ell}$. We have $[\![\mathtt{x}^{\ell}]\!]_{\varphi}^{\overline{\mathtt{CD}}} = \mathtt{x}^{\ell}$ and $[\![\Gamma]\!]_{\varphi}(\mathtt{x}) = \sqcup\varphi(\ell)$, so we can derive, using Rule (17), $[\![\overline{\mathtt{CD}}]\!]_{\varphi}^{\overline{\mathtt{CD}}}; [\![\Gamma]\!]_{\varphi} \vdash [\![\mathtt{e}]\!]_{\varphi}^{\overline{\mathtt{CD}}} \in \sqcup\varphi(\ell)$, as desired.
- (18) $\mathtt{e} \equiv \mathtt{e}_0 . \mathtt{f}_i{}^{\ell}$. We have $\overline{\mathtt{CD}}; \Gamma \vdash \mathtt{e}_0 \in \mathtt{C}_0$ and $fields(\overline{\mathtt{CD}}, \mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$, where $\mathtt{f_i}$ occurs in $\overline{\mathtt{f}}$. From the induction hypothesis we have $[\![\overline{\mathtt{CD}}]\!]_{\varphi}^{\overline{\mathtt{CD}}}; [\![\Gamma]\!]_{\varphi} \vdash [\![\mathtt{e}_0]\!]_{\varphi}^{\overline{\mathtt{CD}}} \in \sqcup\varphi(lab(\mathtt{e}_0))$. From $\varphi \in type\text{-}respecting(\mathtt{P})$ we have $\overline{\mathtt{CD}} \vdash \sqcup\varphi(lab(\mathtt{e}_0))$ <: $\mathtt{C}_0$ so $fields(\overline{\mathtt{CD}}, \sqcup\varphi(lab(\mathtt{e}_0))) = \overline{\mathtt{D}}\ \overline{\mathtt{g}}\ \overline{\mathtt{C}}\ \overline{\mathtt{f}}$. Hence, from Lemma 14 (below), we have $fields([\![\overline{\mathtt{CD}}]\!]_{\varphi}^{\overline{\mathtt{CD}}}, \sqcup\varphi(lab(\mathtt{e}_0))) = \sqcup\varphi(lab(\overline{\mathtt{g}}))\ \overline{\mathtt{g}}\ \sqcup\varphi(lab(\overline{\mathtt{f}}))\ \overline{\mathtt{f}}$. From Rule (26) we

have $\varphi(\mathit{lab}(\mathtt{f_i})) = \varphi(\ell)$, so $\sqcup\varphi(\mathit{lab}(\mathtt{f_i})) = \sqcup\varphi(\ell)$, so we can derive, using Rule (18), that $[\![\overline{\mathtt{CD}}]\!]_\varphi^{\overline{\mathtt{CD}}}; [\![\Gamma]\!]_\varphi \vdash [\![\mathtt{e}]\!]_\varphi^{\overline{\mathtt{CD}}} \in \sqcup\varphi(\ell)$, as desired.

- (19) $\mathtt{e} \equiv \mathtt{new}^\ell\ \mathtt{C}(\overline{\mathtt{e}})$. We have $\mathit{fields}(\overline{\mathtt{CD}}, \mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}}$, and $\overline{\mathtt{CD}}; \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{E}}$, and $\overline{\mathtt{CD}} \vdash \overline{\mathtt{E}}\ \mathtt{<:}\ \overline{\mathtt{D}}$. From Lemma 14 (below) we have $\mathit{fields}([\![\overline{\mathtt{CD}}]\!]_\varphi^{\overline{\mathtt{CD}}}, \mathtt{C}) = \sqcup\varphi(\mathit{lab}(\overline{\mathtt{f}}))\ \overline{\mathtt{f}}$. From the induction hypothesis we have $[\![\overline{\mathtt{CD}}]\!]_\varphi^{\overline{\mathtt{CD}}}; [\![\Gamma]\!]_\varphi \vdash [\![\overline{\mathtt{e}}]\!]_\varphi^{\overline{\mathtt{CD}}} \in \sqcup\varphi(\mathit{lab}(\overline{\mathtt{e}}))$. From Rule (27), we have $\varphi(\mathit{lab}(\overline{\mathtt{e}})) \subseteq \varphi(\mathit{lab}(\overline{\mathtt{f}}))$, so from Lemma 16 (below) we have $\overline{\mathtt{CD}} \vdash \sqcup\varphi(\mathit{lab}(\overline{\mathtt{e}}))\ \mathtt{<:}\ \sqcup\varphi(\mathit{lab}(\overline{\mathtt{f}}))$, and so from Lemma 15 (below) we have $[\![\overline{\mathtt{CD}}]\!]_\varphi^{\overline{\mathtt{CD}}} \vdash \sqcup\varphi(\mathit{lab}(\overline{\mathtt{e}}))\ \mathtt{<:}\ \sqcup\varphi(\mathit{lab}(\overline{\mathtt{f}}))$. Finally we have from Rule (28) that $\mathtt{C} \in \varphi(\ell)$, and since $\mathtt{new}^\ell\ \mathtt{C}(\overline{\mathtt{e}})$ has static type $\mathtt{C}$ and $\varphi \in$ *type-respecting*$(\mathtt{P})$, we have $\sqcup\varphi(\ell) = \mathtt{C}$. We conclude, using Rule (19), that we have $[\![\overline{\mathtt{CD}}]\!]_\varphi^{\overline{\mathtt{CD}}}; [\![\Gamma]\!]_\varphi \vdash \mathtt{new}^\ell\ \mathtt{C}([\![\overline{\mathtt{e}}]\!]_\varphi^{\overline{\mathtt{CD}}}) \in \mathtt{C}$.
- (20) $\mathtt{e} \equiv (\mathtt{C})^\ell \mathtt{e}_0$. We have $\overline{\mathtt{CD}}; \Gamma \vdash \mathtt{e}_0 \in \mathtt{D}$. From the induction hypothesis we have $[\![\overline{\mathtt{CD}}]\!]_\varphi^{\overline{\mathtt{CD}}}; [\![\Gamma]\!]_\varphi \vdash [\![\mathtt{e}_0]\!]_\varphi^{\overline{\mathtt{CD}}} \in \sqcup\varphi(\mathit{lab}(\mathtt{e}_0))$. From Rule (20) we have that we can derive $[\![\overline{\mathtt{CD}}]\!]_\varphi^{\overline{\mathtt{CD}}}; [\![\Gamma]\!]_\varphi \vdash (\sqcup\varphi(\ell))^\ell [\![\mathtt{e}_0]\!]_\varphi^{\overline{\mathtt{CD}}} \in \sqcup\varphi(\ell)$.
- (21) $\mathtt{e} \equiv \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}})^\ell$. There are two cases. First, assume that there is a class $\mathtt{D}$ in $\mathtt{P}$ such that $\forall \mathtt{E} \in \varphi(\mathit{lab}(\mathtt{e}_0)) : \mathit{impl}(\overline{\mathtt{CD}}, \mathtt{E}, \mathtt{m}) = \mathtt{D::m}$. We have

$$\overline{\mathtt{CD}}; \Gamma \vdash \mathtt{e}_0 \in \mathtt{C}_0 \tag{B.1}$$
$$\mathit{mtype}(\overline{\mathtt{CD}}, \mathtt{C}_0, \mathtt{m}) = \overline{\mathtt{D}} \to \mathtt{C} \tag{B.2}$$
$$\overline{\mathtt{CD}}; \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{C}} \tag{B.3}$$

From (B.1), (B.3), and the induction hypothesis, we have

$$[\![\overline{\mathtt{CD}}]\!]_\varphi^{\overline{\mathtt{CD}}}; [\![\Gamma]\!]_\varphi \vdash [\![\mathtt{e}_0]\!]_\varphi^{\overline{\mathtt{CD}}} \in \sqcup\varphi(\mathit{lab}(\mathtt{e}_0)) \tag{B.4}$$
$$[\![\overline{\mathtt{CD}}]\!]_\varphi^{\overline{\mathtt{CD}}}; [\![\Gamma]\!]_\varphi \vdash [\![\overline{\mathtt{e}}]\!]_\varphi^{\overline{\mathtt{CD}}} \in \sqcup\varphi(\mathit{lab}(\overline{\mathtt{e}})) \tag{B.5}$$

We have $\overline{\mathtt{CD}} \vdash \sqcup\varphi(\mathit{lab}(\mathtt{e}_0))\ \mathtt{<:}\ \mathtt{D}$, so from Lemma 15 (below), we have

$$[\![\overline{\mathtt{CD}}]\!]_\varphi^{\overline{\mathtt{CD}}} \vdash \sqcup\varphi(\mathit{lab}(\mathtt{e}_0))\ \mathtt{<:}\ \mathtt{D}, \tag{B.6}$$

and together with (B.2), we have $\mathit{mtype}(\overline{\mathtt{CD}}, \mathtt{D}, \mathtt{m}) = \overline{\mathtt{D}} \to \mathtt{C}$. Suppose also $\mathit{mbody}(\overline{\mathtt{CD}}, \mathtt{D}, \mathtt{m}) = (\ell', \overline{\mathtt{x}}, \mathtt{e}')$. From Lemma 13 (below) we have

$$\mathit{mtype}([\![\overline{\mathtt{CD}}]\!]_\varphi^{\overline{\mathtt{CD}}}, \mathtt{D}, \mathtt{m}) = (\sqcup\varphi(\mathit{lab}(\overline{\mathtt{x}}))) \to (\sqcup\varphi(\ell')). \tag{B.7}$$

From Rule (40) we have $\varphi(\mathit{lab}(\mathtt{e}_0)) \neq \emptyset$, so suppose $\mathtt{E}_0 \in \varphi(\mathit{lab}(\mathtt{e}_0))$. Suppose also $\mathit{mbody}(\overline{\mathtt{CD}}, \mathtt{E}_0, \mathtt{m}) = (\ell'', \overline{\mathtt{x}''}, \mathtt{e}'')$, From Rules (30)–(31) we have

$$\varphi(\mathit{lab}(\overline{\mathtt{e}})) \subseteq \varphi(\mathit{lab}(\overline{\mathtt{x}''}))$$
$$\varphi(\ell'') = \varphi(\ell).$$

Finally, from Rules (38)–(39) we have

$$\varphi(\mathit{lab}(\overline{\mathtt{x}})) \subseteq \varphi(\mathit{lab}(\overline{\mathtt{x}''}))$$
$$\varphi(\ell') = \varphi(\ell''),$$

so

$$\varphi(lab(\overline{\mathtt{e}})) \subseteq \varphi(lab(\overline{\mathtt{x}})) \tag{B.8}$$
$$\varphi(\ell') = \varphi(\ell). \tag{B.9}$$

Thus, from Rule (22), and from (B.4)–(B.9), we have that we can derive

$$[\![\overline{\mathtt{CD}}]\!]_{\varphi}^{\overline{\mathtt{CD}}}; [\![\Gamma]\!]_{\varphi} \vdash [\![\mathtt{e}_0]\!]_{\varphi}^{\overline{\mathtt{CD}}}\mathtt{.D::m(}[\![\overline{\mathtt{e}}]\!]_{\varphi}^{\overline{\mathtt{CD}}}\mathtt{)}^{\ell} \in \sqcup\varphi(\ell)$$

as desired.

Second, suppose we have the "otherwise" case from the definition of the transformation of a method call. The proof of this case is similar to the first case, and we omit the details.

- (22) $\mathtt{e} \equiv \mathtt{e}_0\mathtt{.D::m(}\overline{\mathtt{e}}\mathtt{)}^{\ell}$. The proof of this case is similar to the previous case, and we omit the details.

Lemma 12 *Suppose* $\varphi \in$ *acceptable*$(\mathtt{P}) \cap$ *type-respecting*$(\mathtt{P})$, *and* $\mathtt{P} = (\overline{\mathtt{CD}}, \mathtt{e}_0)$. *If* *can-declare*$(\overline{\mathtt{CD}}, \mathtt{D}, \mathtt{m}, \overline{\mathtt{C}} \rightarrow \mathtt{C}_0)$, $\overline{\mathtt{CD}} \vdash \mathtt{C} <: \mathtt{D}$, *and* *mbody*$(\overline{\mathtt{CD}}, \mathtt{C}, \mathtt{m}) = (\ell, \overline{\mathtt{x}}, \mathtt{e})$, *then*

$$\textit{can-declare}([\![\overline{\mathtt{CD}}]\!]_{\varphi}^{\overline{\mathtt{CD}}}, \mathtt{D}, \mathtt{m}, (\sqcup\varphi(lab(\overline{\mathtt{x}}))) \rightarrow (\sqcup\varphi(\ell))).$$

Proof. Immediate from Lemma 13 (below), using Rule (9), (14)–(16).

Lemma 13 *Suppose* $\varphi \in$ *acceptable*$(\mathtt{P}) \cap$ *type-respecting*$(\mathtt{P})$, *and* $\mathtt{P} = (\overline{\mathtt{CD}}, \mathtt{e}_0)$. *If* *mtype*$(\overline{\mathtt{CD}}, \mathtt{D}, \mathtt{m}) = \overline{\mathtt{D}} \rightarrow \mathtt{D}_0$, $\overline{\mathtt{CD}} \vdash \mathtt{C} <: \mathtt{D}$, *and* *mbody*$(\overline{\mathtt{CD}}, \mathtt{C}, \mathtt{m}) = (\ell, \overline{\mathtt{x}}, \mathtt{e})$, *then*

$$\textit{mtype}([\![\overline{\mathtt{CD}}]\!]_{\varphi}^{\overline{\mathtt{CD}}}, \mathtt{D}, \mathtt{m}) = (\sqcup\varphi(lab(\overline{\mathtt{x}}))) \rightarrow (\sqcup\varphi(\ell)).$$

Proof. Straightforward, using the rules Rules (3)–(6), (14)–(16), (38)–(39).

Lemma 14 *Suppose* $\varphi \in$ *acceptable*$(\mathtt{P}) \cap$ *type-respecting*$(\mathtt{P})$, *and* $\mathtt{P} = (\overline{\mathtt{CD}}, \mathtt{e}_0)$. *If* *fields*$(\overline{\mathtt{CD}}, \mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}}$, *then* *fields*$([\![\overline{\mathtt{CD}}]\!]_{\varphi}^{\overline{\mathtt{CD}}}, \mathtt{D}) = \sqcup\varphi(lab(\overline{\mathtt{g}}))\ \overline{\mathtt{g}}$.

Proof. Straightforward, by induction on the structure of the derivation of the judgment *fields*$(\overline{\mathtt{CD}}, \mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}}$, using Rules (1)–(2).

Lemma 15 *If* $\overline{\mathtt{CD}} \vdash \mathtt{C} <: \mathtt{D}$, *then* $[\![\overline{\mathtt{CD}}]\!]_{\varphi}^{\overline{\mathtt{CD}}} \vdash \mathtt{C} <: \mathtt{D}$.

Proof. Immediate from the definition of subtyping, that is, Rules (14)–(16).

Lemma 16 *Suppose* $S, T \in$ *flow*$(\mathtt{P}) \setminus \emptyset$. *If* $S \subseteq T$, *then* $\overline{\mathtt{CD}} \vdash \sqcup S <: \sqcup T$.

Proof. Immediate from the observation that the subtyping order forms a tree and therefore admits least upper bounds of nonempty sets.

C Proof of Theorem 5

($\Rightarrow$) Let $\mathtt{P}_1 = (\overline{\mathtt{CD}}, \mathtt{X}\langle \mathtt{e}_1 \rangle)$ and $\mathtt{P}_2 = (\overline{\mathtt{CD}}, \mathtt{X}\langle \mathtt{e}_2 \rangle)$ where $\mathtt{e}_1$ and $\mathtt{e}_2$ are given by one of the rules in Figure 2. Clearly $[\![\mathtt{P}_i]\!]_\varphi = ([\![\overline{\mathtt{CD}}]\!]_\varphi^{\overline{\mathtt{CD}}}, [\![\mathtt{X}]\!]_\varphi^{\overline{\mathtt{CD}}}\langle [\![\mathtt{e}_i]\!]_\varphi^{\overline{\mathtt{CD}}} \rangle)$, so it remains to show that $[\![\mathtt{e}_1]\!]_\varphi^{\overline{\mathtt{CD}}} \mapsto [\![\mathtt{e}_2]\!]_\varphi^{\overline{\mathtt{CD}}}$. The interesting cases are when $\mathtt{e}_1$ is a cast and when $\mathtt{e}_1$ is a dynamic method call that is transformed to a static method call.

- Case 1, $\mathtt{e}_1 = (\mathtt{D})^{\ell_1}\mathtt{new}^{\ell_2}\ \mathtt{C}(\overline{\mathtt{e}})$: In this case

$$\mathtt{e}_2 = \mathtt{new}^{\ell_2}\ \mathtt{C}(\overline{\mathtt{e}}) \tag{C.1}$$

$$[\![\mathtt{e}_1]\!]_\varphi^{\overline{\mathtt{CD}}} = (\sqcup\varphi(\ell_1))^{\ell_1}\mathtt{new}^{\ell_2}\ \mathtt{C}([\![\overline{\mathtt{e}}]\!]_\varphi^{\overline{\mathtt{CD}}}) \tag{C.2}$$

$$\overline{\mathtt{CD}} \vdash \mathtt{C} <: \mathtt{D} \tag{C.3}$$

By Rule 28, $\mathtt{C} \in \varphi(\ell_2)$; by Rule 29, $\varphi(\ell_2) \cap \mathit{subclasses}(\mathtt{P}_1, \mathtt{D}) \subseteq \varphi(\ell_1)$. Thus $\mathtt{C} \in \varphi(\ell_1)$, so $\overline{\mathtt{CD}} \vdash \mathtt{C} <: \sqcup\varphi(\ell_1)$. By the reduction rules,

$$[\![\mathtt{e}_1]\!]_\varphi^{\overline{\mathtt{CD}}} \mapsto \mathtt{new}^{\ell_2}\ \mathtt{D}([\![\overline{\mathtt{e}}]\!]_\varphi^{\overline{\mathtt{CD}}}) = [\![\mathtt{e}_2]\!]_\varphi^{\overline{\mathtt{CD}}} \tag{C.4}$$

- Case 2, $\mathtt{e}_1 = \mathtt{new}^{\ell_1}\ \mathtt{C}(\overline{\mathtt{e}}).\mathtt{m}(\overline{\mathtt{d}})^{\ell_2}$ and $[\![\mathtt{e}_1]\!]_\varphi^{\overline{\mathtt{CD}}} = \mathtt{new}^{\ell_1}\ \mathtt{C}([\![\overline{\mathtt{e}}]\!]_\varphi^{\overline{\mathtt{CD}}}).\mathtt{D::m}([\![\overline{\mathtt{d}}]\!]_\varphi^{\overline{\mathtt{CD}}})^{\ell_2}$: In this case

$$\forall \mathtt{E} \in \varphi(\ell_1) : \mathit{impl}(\overline{\mathtt{CD}}, \mathtt{E}, \mathtt{m}) = \mathtt{D::m} \tag{C.5}$$

$$\mathtt{e}_2 = \mathtt{e}_0\{\mathtt{this}, \overline{\mathtt{x}} := \mathtt{new}^{\ell_1}\ \mathtt{C}(\overline{\mathtt{e}}), \overline{\mathtt{d}}\} \tag{C.6}$$

where

$$\mathit{mbody}(\overline{\mathtt{CD}}, \mathtt{C}, \mathtt{m}) = (\ell, \overline{\mathtt{x}}, \mathtt{e}_0) \tag{C.7}$$

By Rule 28, $\mathtt{C} \in \varphi(\ell_1)$. By (C.5), $\mathit{impl}(\overline{\mathtt{CD}}, \mathtt{C}, \mathtt{m}) = \mathtt{D::m}$. By inspecting Rules 5–8 and (C.7)

$$\mathit{mbody}(\overline{\mathtt{CD}}, \mathtt{D}, \mathtt{m}) = (\ell, \overline{\mathtt{x}}, \mathtt{e}_0) \tag{C.8}$$

Thus:

$$\begin{aligned}
[\![\mathtt{e}_1]\!]_\varphi^{\overline{\mathtt{CD}}} &= \mathtt{new}^{\ell_1}\ \mathtt{C}([\![\overline{\mathtt{e}}]\!]_\varphi^{\overline{\mathtt{CD}}}).\mathtt{D::m}([\![\overline{\mathtt{d}}]\!]_\varphi^{\overline{\mathtt{CD}}})^{\ell_2} \\
&\mapsto [\![\mathtt{e}_0]\!]_\varphi^{\overline{\mathtt{CD}}}\{\mathtt{this}, \overline{\mathtt{x}} := \mathtt{new}^{\ell_1}\ \mathtt{C}([\![\overline{\mathtt{e}}]\!]_\varphi^{\overline{\mathtt{CD}}}), [\![\overline{\mathtt{d}}]\!]_\varphi^{\overline{\mathtt{CD}}}\} \\
&= [\![\mathtt{e}_2]\!]_\varphi^{\overline{\mathtt{CD}}}
\end{aligned}$$

($\Leftarrow$) If $[\![\mathtt{P}_1]\!]_\varphi$ takes any step then it is easy to see that $\mathtt{P}_1$ has the form $(\overline{\mathtt{CD}}, \mathtt{X}\langle \mathtt{e}_1 \rangle)$ and that $[\![\mathtt{P}_1]\!]_\varphi \mapsto ([\![\overline{\mathtt{CD}}]\!]_\varphi^{\overline{\mathtt{CD}}}, [\![\mathtt{X}]\!]_\varphi^{\overline{\mathtt{CD}}}\langle \mathtt{e}' \rangle)$ for $[\![\mathtt{e}_1]\!]_\varphi^{\overline{\mathtt{CD}}}$ and $\mathtt{e}'$ as in the rules in Figure 2. It remains to show that $\mathtt{e}_1 \mapsto \mathtt{e}_2$ and $[\![\mathtt{e}_2]\!]_\varphi^{\overline{\mathtt{CD}}} = \mathtt{e}'$ for some $\mathtt{e}_2$. The interesting cases are when $\mathtt{e}_1$ is a cast and when $\mathtt{e}_1$ is a dynamic method call that is transformed to a static method call.

- Case 1, $\mathtt{e}_1 = \mathtt{(D)}^{\ell_1}\mathtt{new}^{\ell_2}\ \mathtt{C}(\overline{\mathtt{e}})$: In this case

$$\llbracket \mathtt{e}_1 \rrbracket_{\varphi}^{\overline{\mathtt{CD}}} = (\sqcup\varphi(\ell_1))^{\ell_1}\mathtt{new}^{\ell_2}\ \mathtt{C}(\llbracket \overline{\mathtt{e}} \rrbracket_{\varphi}^{\overline{\mathtt{CD}}}) \tag{C.9}$$

$$\mathtt{e}' = \mathtt{new}^{\ell_2}\ \mathtt{C}(\llbracket \overline{\mathtt{e}} \rrbracket_{\varphi}^{\overline{\mathtt{CD}}}) \tag{C.10}$$

$$\overline{\mathtt{CD}} \vdash \mathtt{C} \mathrel{\mathtt{<:}} \sqcup\varphi(\ell_1) \tag{C.11}$$

Since φ is type respecting and Rule 20, $\overline{\mathtt{CD}} \vdash \sqcup\varphi(\ell_1) \mathrel{\mathtt{<:}} \mathtt{D}$. By transitivity of subtyping, $\overline{\mathtt{CD}} \vdash \mathtt{C} \mathrel{\mathtt{<:}} \mathtt{D}$. Then $\mathtt{e}_1 \mapsto \mathtt{e}_2$ and $\llbracket \mathtt{e}_2 \rrbracket_{\varphi}^{\overline{\mathtt{CD}}} = \mathtt{e}'$ if $\mathtt{e}_2$ is $\mathtt{new}^{\ell_2}\ \mathtt{C}(\overline{\mathtt{e}})$.

- Case 2, $\mathtt{e}_1$=$\mathtt{new}^{\ell_1}\ \mathtt{C}(\overline{\mathtt{e}}).\mathtt{m}(\overline{\mathtt{d}})^{\ell_2}$ and $\llbracket \mathtt{e}_1 \rrbracket_{\varphi}^{\overline{\mathtt{CD}}} = \mathtt{new}^{\ell_1}\ \mathtt{C}(\llbracket \overline{\mathtt{e}} \rrbracket_{\varphi}^{\overline{\mathtt{CD}}}).\mathtt{D::m}(\llbracket \overline{\mathtt{d}} \rrbracket_{\varphi}^{\overline{\mathtt{CD}}})^{\ell_2}$: In this case

$$\forall \mathtt{E} \in \varphi(\ell_1) : \mathit{impl}(\overline{\mathtt{CD}}, \mathtt{E}, \mathtt{m}) = \mathtt{D::m} \tag{C.12}$$

$$\mathtt{e}' = \llbracket \mathtt{e}_0 \rrbracket_{\varphi}^{\overline{\mathtt{CD}}}\{\mathtt{this}, \overline{\mathtt{x}} := \mathtt{new}^{\ell_1}\ \mathtt{C}(\llbracket \overline{\mathtt{e}} \rrbracket_{\varphi}^{\overline{\mathtt{CD}}}), \llbracket \overline{\mathtt{d}} \rrbracket_{\varphi}^{\overline{\mathtt{CD}}}\} \tag{C.13}$$

where

$$\mathit{mbody}(\overline{\mathtt{CD}}, \mathtt{D}, \mathtt{m}) = (\ell, \overline{\mathtt{x}}, \mathtt{e}_0) \tag{C.14}$$

By Rule 28, $\mathtt{C} \in \varphi(\ell_1)$. By (C.12), $\mathit{impl}(\overline{\mathtt{CD}}, \mathtt{C}, \mathtt{m}) = \mathtt{D::m}$. By inspection of Rules 5–8 and (C.14)

$$\mathit{mbody}(\overline{\mathtt{CD}}, \mathtt{C}, \mathtt{m}) = (\ell, \overline{\mathtt{x}}, \mathtt{e}_0) \tag{C.15}$$

Then $\mathtt{e}_1 \mapsto \mathtt{e}_2$ and $\llbracket \mathtt{e}_2 \rrbracket_{\varphi}^{\overline{\mathtt{CD}}} = \mathtt{e}'$ if $\mathtt{e}_2$ is

$$\mathtt{e}_0\{\mathtt{this}, \overline{\mathtt{x}} := \mathtt{new}^{\ell_1}\ \mathtt{C}(\overline{\mathtt{e}}), \overline{\mathtt{d}}\} \tag{C.16}$$

References

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[2] J. Gosling, B. Joy, G. Steele, The Java Language Specification, Addison-Wesley, 1996.

[3] M. A. Ellis, B. Stroustrup, The Annotated C++ Reference Manual, Addison-Wesley, 1990.

[4] T. Lindholm, F. Yellin, The Java Virtual Machine Specification, Addison-Wesley, 1996.

[5] G. Morrisett, D. Tarditi, P. Cheng, C. Stone, R. Harper, P. Lee, The TIL/ML compiler: Performance and safety through types, in: ACM SIGPLAN Workshop on Compiler Support for System Software, Tucson, AZ, USA, 1996.

[6] G. Morrisett, D. Walker, K. Crary, N. Glew, From System F to typed assembly language, ACM Transations on Programming Languages and Systems 21 (3) (1999) 528–569.

[7] A. Wright, S. Jagannathan, C. Ungureanu, A. Hertzmann, Compiling Java to a typed lambda-calculus: A preliminary report, in: ACM Workshop on Types in Compilation, Kyoto, Japan, 1998, pp. 9–27.

[8] N. Glew, An efficient class and object encoding, in: Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), Minneapolis, Minnesota, USA, 2000, pp. 311–324.

[9] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, P. Lee, TIL: A type-directed optimizing compiler for ML, in: 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation, Philadelphia, PA, USA, 1996, pp. 181–192.

[10] F. Tip, A. Kiezun, D. Bäumer, Refactoring for generalization using type constraints, manuscript (2003).

[11] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, T. Nakatani, A study of devirtualization techniques for a Java just-in-time compiler, in: Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), Minneapolis, Minnesota, USA, 2000, pp. 294–310.

[12] J. Dean, D. Grove, C. Chambers, Optimization of object-oriented programs using static class hierarchy analysis, in: W. Olthoff (Ed.), Ninth European Conference on Object-Oriented Programming (ECOOP), Vol. 952 of Lecture Notes in Computer Science, Springer-Verlag, Aarhus, Denmark, 1995, pp. 77–101.

[13] D. Detlefs, O. Agesen, Inlining of virtual methods, in: Thirteenth European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag (LNCS 1628), 1999, pp. 258–278.

[14] E. M. Gagnon, L. J. Hendren, G. Marceau, Efficient inference of static types for Java bytecode, in: Seventh International Static Analysis Symposium (SAS), Springer-Verlag (*LNCS* 1824), 2000, pp. 199–219.

[15] J. Palsberg, M. I. Schwartzbach, Object-Oriented Type Systems, John Wiley & Sons, 1994.

[16] T. Knoblock, J. Rehof, Type elaboration and subtype completion for Java Bytecode, ACM Transations on Programming Languages and Systems 23 (2) (2001) 243–272.

[17] J. Palsberg, P. M. O'Keefe, A type system equivalent to flow analysis, ACM Transations on Programming Languages and Systems 17 (4) (1995) 576–599, preliminary version in Proceedings of POPL'95.

[18] N. Heintze, Control-flow analysis and type systems, in: Second International Static Analysis Symposium (SAS), Springer-Verlag (*LNCS* 983), Glasgow, Scotland, 1995, pp. 189–206.

[19] J. Palsberg, Equality-based flow analysis versus recursive types, ACM Transations on Programming Languages and Systems 20 (6) (1998) 1251–1264.

[20] J. Palsberg, C. Pavlopoulou, From polyvariant flow information to intersection and union types, Journal of Functional Programming 11 (3) (2001) 263–317, preliminary version in Proceedings of POPL'98.

[21] L. O. Andersen, Self-applicable C program specialization, in: 1992 Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM), 1992, pp. 54–61, (Technical Report YALEU/DCS/RR-909, Yale University).

[22] J. Palsberg, Closure analysis in constraint form, ACM Transations on Programming Languages and Systems 17 (1) (1995) 47–62, preliminary version in Proceedings of CAAP'94.

[23] M. Fähndrich, A. Aiken, Program analysis using mixed term and set constraints, in: Fourth International Static Analysis Symposium (SAS), Springer-Verlag (*LNCS*), 1997, pp. 114–126.

[24] N. Glew, J. Palsberg, Method inlining, dynamic class loading, and type soundness, submitted for publication (2003).

[25] A. Igarashi, B. Pierce, P. Wadler, Featherweight Java: A minimal core calculus for Java and GJ, ACM Transations on Programming Languages and Systems 23 (3) (2001) 396–450, first appeared in OOPSLA 1999.

[26] F. Nielson, The typed lambda-calculus with first-class processes, in: Proceedings of PARLE'89, 1989, pp. 357–373.

[27] A. Wright, M. Felleisen, A syntactic approach to type soundness, Information and Computation 115 (1) (1994) 38–94.

[28] S. Jagannathan, A. Wright, S. Weeks, Type-directed flow analysis for typed intermediate languages, in: Fourth International Static Analysis Symposium (SAS), Springer-Verlag (*LNCS*), 1997, pp. 232–249.

[29] D. F. Bacon, P. F. Sweeney, Fast static analysis of C++ virtual function calls, in: Eleventh Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), San Jose, CA, 1996, pp. 324–341, *SIGPLAN Notices* 31(10).

[30] D. F. Bacon, Fast and effective optimization of statically typed object-oriented languages, Ph.D. thesis, Computer Science Division, University of California, Berkeley, report No. UCB/CSD-98-1017 (Dec. 1997).

[31] F. Tip, J. Palsberg, Scalable propagation-based call graph construction algorithms, in: Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), Minneapolis, Minnesota, USA, 2000, pp. 281–293.

[32] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, C. Godin, Practical virtual method call resolution for Java, in: Proceedings of the Fifteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00), Minneapolis, Minnesota), 2000, pp. 264–280.