Type-Safe Method Inlining

Neal Glew[1] and Jens Palsberg[2]

[1] aglew@acm.org
[2] Purdue University, Dept of Computer Science, W Lafayette, IN 47907, USA, palsberg@cs.purdue.edu.

Abstract. In a typed language such as Java, inlining of virtual methods does not always preserve typability. While inlining has been intensively studied for more than a decade, 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; all in the context of a variant of Featherweight Java.

1 Introduction

1.1 Background

A number of recent compilers use typed intermediate languages (*e.g.*, [11, 12, 22, 6]) to obtain debugging and optimisation benefits [11, 20]. Such compilers require transformations that are type safe, that is, transformations that produce output that type checks. This paper is concerned with one such transformation, that of inlining of method calls in a statically-typed object-oriented language. While there has been substantial previous work on method inlining (*e.g.*, [2, 3]), the known approaches are either for an untyped language, or have to rely on adding type casts or extra types, as we explain next.

Consider the following well-typed Java program:

```
class B {
  void m() {}
}

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

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

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

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

```
x.f = x;                              // does not type check
((B)new C()).f = ((B)new C());        // 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()`. Moreover, even if there was an `f`-field of type `C`, then the right-hand side of each assignment is not of type `C` or a subtype of `C`.

The problem can be solved by inserting type casts. In their Java compiler, Wright, Jagannathan, Ungureanu, and Hertzmann [22] 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`:

```
void m() {
  ((C)this).f = (C)this;
}
```

After inlining, the two assignments type check:

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

A different approach was taken by Gagnon, Hendren, and Marceau [5] who first compile Java to an untyped representation of Java bytecode, and then infer types to regain static type annotations. 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 third approach, which does not require type casts at all, is to add new types to the program. Knoblock and Rehof [10] demonstrated how to do that automatically in such a way 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. Until now, it has remained an open problem to devise a scheme for supporting type-safe method inlining in a way that does not require the insertion of type casts or new types.

1.2 Our Result

We present an approach to type-safe method inlining that never hurts performance, and does not require the insertion of type casts or new types. Our approach 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. We present the transformation, the conditions on the flow analysis, and prove the correctness properties; all in the context of a variant of Featherweight Java.

The flow analysis 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. Given such a set the transformation can use the least upper bound as the new explicit type information. For our example program, the best flow set for both receiver expressions in the program is {`C`}, and the least upper bound for this set is `C`. Therefore, in the transformed program the code snippet is as follows—and it type checks:

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

Our flow analyses must satisfy a mixture of subset constraints [1, 14] and equality constraints [19, 15]. Fähndrich and Aiken [4] studied this idea. We need both kinds of constraints in order to align the flow analysis with the Java type system. It is well known that subsets in a flow analysis are closely related to subtyping in type systems, and that for a variety of cases it is possible to match up flow analyses with type systems in a strong sense [16, 7, 15, 17, 10]. The Java type system allows the use of subtyping in some places but not in others. Thus, our flow analyses must satisfy subset constraints in some places and equality constraints in others.

Our approach does method inlining in two steps:

1. change some dynamic method invocations to static method invocations (and change the type annotations and the classes used in type casts) and
2. inline the static method invocations.

The idea is, as in previous work, that in a dynamic method dispatch `e.m(`e_1`, ..., `e_n`)`, if all the objects that `e` could evaluate to are instances of classes which inherit `m` from a fixed class `D`, then the dynamic dispatch can be transformed to a static dispatch `e.D::m(`e_1`, ..., `e_n`)`.[1] A static dispatch `e.D::m(`e_1`, ..., `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 for method `m` body `e'` and parameters $\mathtt{x}_1$ through $\mathtt{x}_n$. This is nothing other than applying a nonstandard reduction rule at compile time, and it is straightforward to show that the rule is type safe.

[1] The expression `e.D::m(`e_1`, ..., `e_n`)` invokes `D`'s version of `m` on `e` with $\mathtt{e}_1$ through $\mathtt{e}_n$ as arguments.

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

2 The Language

We formalise our results in Featherweight Java [8] (FJ) extended with a static dispatch 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 invocation 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 invocation, and static method invocation. Although FJS does not have `super`, static method invocation 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:

$$
\begin{array}{lcl}
\mathtt{P} & ::= & (\overline{\mathtt{CD}},\mathtt{e}) \\
\mathtt{CD} & ::= & \mathtt{class\ C\ extends\ C\ \{}\overline{\mathtt{C}}\ \overline{\mathtt{f}^{\ell}}\mathtt{;\ K\ }\overline{\mathtt{M}}\mathtt{\}} \\
\mathtt{K} & ::= & \mathtt{C(}\overline{\mathtt{C}}\ \overline{\mathtt{f}}\mathtt{)\ \{super(}\overline{\mathtt{f}}\mathtt{);\ this.}\overline{\mathtt{f}}\mathtt{\ =\ }\overline{\mathtt{f}}\mathtt{;\}} \\
\mathtt{M} & ::= & \mathtt{C\ m(}\overline{\mathtt{C}}\ \overline{\mathtt{x}^{\ell}}\mathtt{)\ \{return}^{\ell}\mathtt{e;\}} \\
\mathtt{e} & ::= & \mathtt{x}^{\ell} \\
 & | & \mathtt{e.f}^{\ell} \\
 & | & \mathtt{new}^{\ell}\ \mathtt{C(}\overline{\mathtt{e}}\mathtt{)} \\
 & | & \mathtt{(C)}^{\ell}\mathtt{e} \\
 & | & \mathtt{e.m(}\overline{\mathtt{e}}\mathtt{)}^{\ell} \\
 & | & \mathtt{e.C::m(}\overline{\mathtt{e}}\mathtt{)}^{\ell}
\end{array}
$$

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 on constructors; `CD` ranges over class definitions; and `P` ranges over programs. `Object` is a class name, but no program may give it a definition; $\mathtt{this}$ is a variable, but no program may use it as a parameter. The over bar notation denotes sequences, so $\overline{\mathtt{f}}$ abbreviates $\mathtt{f}_1,\ \ldots,\ \mathtt{f}_n$. This notation also denotes pairs of sequences in an obvious way—$\overline{\mathtt{C}}\ \overline{\mathtt{f}^{\ell}}$ abbreviates $\mathtt{C}_1\ \mathtt{f}_1^{\ell_1},\ \ldots,\ \mathtt{C}_n\ \mathtt{f}_n^{\ell_n}$, $\overline{\mathtt{C}}\ \overline{\mathtt{f}^{\ell}};$ abbreviates $\mathtt{C}_1\ \mathtt{f}_1^{\ell_1};\ \cdots;\ \mathtt{C}_n\ \mathtt{f}_n^{\ell_n};$, and $\mathtt{this}.\overline{\mathtt{f}}\mathtt{=}\overline{\mathtt{f}}$ abbreviates $\mathtt{this.f}_1\mathtt{=f}_1;\ \cdots;\ \mathtt{this.f}_n\mathtt{=f}_n$. The empty sequence is $\bullet$, 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 $\overline{\mathtt{CD}}(\mathtt{C})$ to mean the definition of $\mathtt{C}$ under the map corresponding to $\overline{\mathtt{CD}}$. Any class name $\mathtt{C}$ except $\mathtt{Object}$ appearing in a program must be given a definition by that program, and the $\mathtt{extends}$ clauses of a program must be acyclic.

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

Metavariable ℓ ranges over a set of labels. Notice that there is a label associated with all expressions, fields, methods, and formal arguments; these labels are not assumed to be unique. For a program $\mathtt{P}$, *labels*($\mathtt{P}$) denotes the set of labels used in $\mathtt{P}$. 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 $\mathtt{this}$ in a class must have the same label. Any well-typed program can easily 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 the FJ paper, we do not make the list of class declarations global, but have them appear explicitly as parameters to functions, predicates, and rules. Function $\mathit{fields}(\overline{\mathtt{CD}},\mathtt{C})$ returns a list of $\mathtt{C}$'s fields and their types; $\mathit{mtype}(\overline{\mathtt{CD}},\mathtt{C},\mathtt{m})$ returns the type of method $\mathtt{m}$ in class $\mathtt{C}$, this type has the form $\overline{\mathtt{C}} \to \mathtt{C}$ where $\mathtt{C}$ is the return type and $\overline{\mathtt{C}}$ are the argument types; $\mathit{mbody}(\overline{\mathtt{CD}},\mathtt{C},\mathtt{m})$ returns the body of method $\mathtt{m}$ in class $\mathtt{C}$, this has the form $(\ell,\overline{\mathtt{x}},\mathtt{e})$ where ℓ is the label of the return statement, $\mathtt{e}$ 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 $\mathtt{C}$ inherits method $\mathtt{m}$ (this might be $\mathtt{C}$ itself if $\mathtt{C}$ declares $\mathtt{m}$), this has the form $\mathtt{D::m}$ where $\mathtt{D}$ is the class. Predicate $\mathit{override}(\overline{\mathtt{CD}},\mathtt{D},\mathtt{m},\overline{\mathtt{C}} \to \mathtt{C})$ is true when method $\mathtt{m}$ of type $\overline{\mathtt{C}} \to \mathtt{D}$ may be declared in a subclass of $\mathtt{D}$. It checks that if $\mathtt{D}$ declares or inherits $\mathtt{m}$ 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).

Field Lookup:

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

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

Method Type Lookup:

$$\frac{\overline{\mathtt{CD}}(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{}\overline{\mathtt{C}\ \mathtt{f}^{\ell_1}}\mathtt{;\ K\ }\overline{\mathtt{M}}\mathtt{\}} \qquad \mathtt{B}_0\ \mathtt{m}(\overline{\mathtt{B}\ \mathtt{x}^{\ell_2}})\ \mathtt{\{return}^{\ell}\mathtt{e;\}} \in \overline{\mathtt{M}}}{\mathit{mtype}(\overline{\mathtt{CD}}, \mathtt{C}, \mathtt{m}) = \overline{\mathtt{B}} \to \mathtt{B}_0} \tag{3}$$

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

Method Body Lookup:

$$\frac{\overline{\mathtt{CD}}(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{}\overline{\mathtt{C}\ \mathtt{f}^{\ell_1}}\mathtt{;\ K\ }\overline{\mathtt{M}}\mathtt{\}} \qquad \mathtt{B}_0\ \mathtt{m}(\overline{\mathtt{B}\ \mathtt{x}^{\ell_2}})\ \mathtt{\{return}^{\ell}\mathtt{e;\}} \in \overline{\mathtt{M}}}{\mathit{mbody}(\overline{\mathtt{CD}}, \mathtt{C}, \mathtt{m}) = (\ell, \overline{\mathtt{x}}, \mathtt{e})} \tag{5}$$

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

Class of Method Lookup:

$$\frac{\overline{\mathtt{CD}}(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{}\overline{\mathtt{C}\ \mathtt{f}^{\ell_1}}\mathtt{;\ K\ }\overline{\mathtt{M}}\mathtt{\}} \qquad \mathtt{B}_0\ \mathtt{m}(\overline{\mathtt{B}\ \mathtt{x}^{\ell_2}})\ \mathtt{\{return}^{\ell}\mathtt{e;\}} \in \overline{\mathtt{M}}}{\mathit{impl}(\overline{\mathtt{CD}}, \mathtt{C}, \mathtt{m}) = \mathtt{C::m}} \tag{7}$$

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

Valid Method Overriding:

$$\frac{\mathit{mtype}(\overline{\mathtt{CD}}, \mathtt{D}, \mathtt{m}) = \overline{\mathtt{D}} \to \mathtt{D}_0 \text{ implies } \overline{\mathtt{C}} = \overline{\mathtt{D}} \text{ and } \mathtt{C}_0 = \mathtt{D}_0}{\mathit{override}(\overline{\mathtt{CD}}, \mathtt{D}, \mathtt{m}, \overline{\mathtt{C}} \to \mathtt{C}_0)} \tag{9}$$

Fig. 1. Auxiliary Definitions

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

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

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

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

Fig. 2. Operational Semantics

The operational semantics of the language appear in Figure 2. Metavariable $\mathtt{X}$ 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 $\mathtt{X}$ by the expression $\mathtt{e}$. Unlike the FJ paper, in addition to making the list of class declarations explicit in the rules we make the evaluation context explicit as well.

Because the language is functional and each class has exactly one constructor of a particular form, the values of the language, which are all objects, can be represented using object constructors $\mathtt{new}^{\ell}\ \mathtt{C}(\overline{\mathtt{e}})$. 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 $\mathtt{D}$ is a subclass of $\mathtt{C}$. If $\mathtt{D}$ is not a subclass of $\mathtt{C}$ then the cast is irreducible representing that the cast fails as a checked run-time error. The reduction rule for method invocation $\mathtt{new}^{\ell_1}\ \mathtt{C}(\overline{\mathtt{e}}).\mathtt{m}(\overline{\mathtt{d}})^{\ell_2}$ looks up the method body of $\mathtt{m}$ in $\mathtt{C}$, if this is $(\ell, \overline{\mathtt{x}}, \mathtt{e}_0)$ then the reduced expression is the body $\mathtt{e}_0$ with the actuals $\overline{\mathtt{d}}$ substituted for the formals $\overline{\mathtt{x}}$ and the object $\mathtt{new}^{\ell_1}\ \mathtt{C}(\overline{\mathtt{e}})$ substituted for `this`. Static method invocation $\mathtt{new}^{\ell_1}\ \mathtt{C}(\overline{\mathtt{e}}).\mathtt{D}{::}\mathtt{m}(\overline{\mathtt{d}})^{\ell}$ reduces similarly except that the method is looked up in $\mathtt{D}$ not $\mathtt{C}$. Note that this method lookup can be done at compile time and a static method invocation can be implemented as a direct call rather than an indirect call through a virtual-dispatch table.

An irreducible expression is *stuck* if it is of the form $\mathtt{X}\langle\mathtt{e.f}^{\ell}\rangle$, $\mathtt{X}\langle\mathtt{e.m}(\overline{\mathtt{e}})^{\ell}\rangle$, or $\mathtt{X}\langle\mathtt{e.D}{::}\mathtt{m}(\overline{\mathtt{e}})^{\ell}\rangle$. The type system prevents stuck expressions from occurring during execution of a program. Irreducible expressions that are not stuck are of the form $\mathtt{v}{::=}\mathtt{new}^{\ell}\ \mathtt{C}(\overline{\mathtt{v}})$ or $\mathtt{X}\langle(\mathtt{C})^{\ell_1}\mathtt{new}^{\ell_2}\ \mathtt{D}(\overline{\mathtt{e}})\rangle$ where $\mathtt{D}$ is not a subclass of $\mathtt{C}$; 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
$\overline{\mathtt{CD}} \vdash \mathtt{C} <: \mathtt{D}$	$\mathtt{C}$ is a subtype of $\mathtt{D}$
$\overline{\mathtt{CD}}; \Gamma \vdash \mathtt{e} \in \mathtt{C}$	$\mathtt{e}$ is well formed and of type $\mathtt{C}$
$\overline{\mathtt{CD}} \vdash \mathtt{M}$ OK in $\mathtt{C}$	$\mathtt{M}$ is well formed in class $\mathtt{C}$
$\overline{\mathtt{CD}} \vdash \mathtt{CD}$ OK	$\mathtt{CD}$ is well formed
$\vdash \mathtt{P} \in \mathtt{C}$	$\mathtt{P}$ is well formed and of type $\mathtt{C}$

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

The rules for constructors and method invocation check that each actual has a subtype of the corresponding formal. The typing rule for dynamic method dispatch looks up the type of the method in the class of the receiver. The typing rule for static method dispatch $\mathtt{e.D}{::}\mathtt{m}(\overline{\mathtt{e}})^{\ell}$ requires that $\mathtt{e}$ has some subtype of $\mathtt{D}$ and looks up the type of the method in $\mathtt{D}$. As in FJ, the typing rules allow

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:

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

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

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

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

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

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

Method Typing:

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

Class Typing:

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

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

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 in 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 methods [13, 21, 8].

Theorem 1 (Type Soundness). *If* $\vdash \texttt{P} \in \texttt{C}$ *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 instances 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 $\textit{classes}(\texttt{P})$; $\textit{subclasses}(\texttt{P}, \texttt{C})$ is the set of subclasses of C (including C). Flow information for P is a member of $\textit{flow-information}(\texttt{P}) = \textit{labels}(\texttt{P}) \rightarrow \textit{flow}(\texttt{P})$. Metavariables S and T range over $\textit{flow}(\texttt{P})$ and 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, Wright, and Weeks [9].) Intuitively, an acceptable analysis contains sets that are *big* enough, in that it correctly approximate 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:

$$\begin{aligned}
\textit{acceptable}(\texttt{P}) &= \{\ \varphi \in \textit{flow-information}(\texttt{P}) \ \mid \\
&\qquad \varphi \text{ satisfies the conditions listed in Figure 4 } \} \\
\textit{type-respecting}(\texttt{P}) &= \{\ \varphi \in \textit{flow-information}(\texttt{P}) \ \mid \\
&\qquad \forall \ell \in \textit{labels}(\texttt{P}) : \varphi(\ell) \subseteq \textit{subclasses}(\texttt{P}, \textit{static-type}(\ell, \texttt{P}))\ \} \\
\textit{flow-analysis}(\texttt{P}) &= \textit{acceptable}(\texttt{P}) \cap \textit{type-respecting}(\texttt{P}).
\end{aligned}$$

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).

First, notice that Rule (40) ensures that least upper bounds are of a nonempty set. Rules (26)–(37) are related to one way of specifying 0-CFA [18, 14, 15]. The unusual aspect of them is that they are a mixture of subset constraints [14] and equality constraints [15]. 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 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 calls, Rule (33) for static 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). Rules (29), (32), and (35) have no counter parts in the type system and are needed to ensure that the flow analysis approximates the results of evaluating expressions. 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.

A variant of Class Hierarchy Analysis [2] (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 $\mathit{flow\text{-}analysis}(\mathtt{P})$.*

Note that $\mathit{flow\text{-}analysis}(\mathtt{P})$ does not have a least element. This is due to Rule (40) that requires all flows to be nonempty. 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. Then $\mathit{flow\text{-}analysis}(\mathtt{P})$ 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. A similar result was proved for λ-calculus [14].

Theorem 3 (Flow Preservation). *If $\varphi \in \mathit{flow\text{-}analysis}(\mathtt{P}_1)$ and $\mathtt{P}_1 \mapsto \mathtt{P}_2$, then $\varphi \in \mathit{flow\text{-}analysis}(\mathtt{P}_2)$.*

– for each $\mathtt{e.f}^{\ell}$ in $\mathtt{P}$:

$$\varphi(lab(\mathtt{f})) = \varphi(\ell) \tag{26}$$

– for each $\mathtt{new}^{\ell}\ \mathtt{D}(\overline{\mathtt{e}})$ in $\mathtt{P}$, where $\mathit{fields}(\overline{\mathtt{CD}}, \mathtt{D}) = \overline{\mathtt{E}}\ \overline{\mathtt{f}}$:

$$\varphi(lab(\overline{\mathtt{e}})) \subseteq \varphi(lab(\overline{\mathtt{f}})) \tag{27}$$

$$\mathtt{D} \in \varphi(\ell) \tag{28}$$

– for each $(\mathtt{D})^{\ell}\mathtt{e}$ in $\mathtt{P}$:

$$\varphi(lab(\mathtt{e})) \cap \mathit{subclasses}(\mathtt{P}, \mathtt{D}) = \varphi(\ell) \tag{29}$$

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

$$\mathtt{D} \in \varphi(lab(\mathtt{e})) \Rightarrow \varphi(lab(\overline{\mathtt{e}})) \subseteq \varphi(lab(\overline{\mathtt{x}})) \tag{30}$$

$$\mathtt{D} \in \varphi(lab(\mathtt{e})) \Rightarrow \varphi(\ell') = \varphi(\ell) \tag{31}$$

$$\varphi(lab(\mathtt{e})) \subseteq \varphi(\ell'') \tag{32}$$

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

$$\varphi(lab(\overline{\mathtt{e}})) \subseteq \varphi(lab(\overline{\mathtt{x}})) \tag{33}$$

$$\varphi(\ell') = \varphi(\ell) \tag{34}$$

$$\varphi(lab(\mathtt{e})) \subseteq \varphi(\ell'') \tag{35}$$

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

$$\mathtt{C} \in \varphi(\ell) \tag{36}$$

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

$$\varphi(lab(\mathtt{e}_0) \subseteq \varphi(\ell) \tag{37}$$

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

$$\overline{\mathtt{CD}}(\mathtt{C}) = \mathtt{class\ C\ extends\ D\ \{}\overline{\mathtt{C}}\ \overline{\mathtt{f}}\mathtt{;\ K}\ \overline{\mathtt{M}}\mathtt{\}}$$

$$\mathit{mbody}(\overline{\mathtt{CD}}, \mathtt{C}, \mathtt{m}) = (\ell_1, \overline{\mathtt{x}}_1, \mathtt{e}_1)$$

$$\mathit{mbody}(\overline{\mathtt{CD}}, \mathtt{D}, \mathtt{m}) = (\ell_2, \overline{\mathtt{x}}_2, \mathtt{e}_2)$$

then

$$\varphi(lab(\overline{\mathtt{x}}_1)) = \varphi(lab(\overline{\mathtt{x}}_2)) \tag{38}$$

$$\varphi(\ell_1) = \varphi(\ell_2) \tag{39}$$

– for each $\ell \in \mathit{labels}(\mathtt{P})$:

$$\varphi(\ell) \neq \emptyset \tag{40}$$

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

It is straightforward to compute $CHA(\mathtt{P})$. However, since $CHA(\mathtt{P})$ is the greatest element of $\mathit{flow\text{-}analysis}(\mathtt{P})$, 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 $\mathit{flow\text{-}analysis}(\mathtt{P})$ does not have a least element so there is not a unique best choice of flow analysis that improves on $CHA(\mathtt{P})$.

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

$$\begin{aligned}
\mathit{lift}(\varphi) &: \mathit{flow\text{-}information}(\mathtt{P}) \rightarrow \mathit{flow\text{-}information}(\mathtt{P}) \\
\mathit{lift}(\varphi) &= \lambda \ell. \begin{cases} \{\ \mathit{static\text{-}type}(\ell, \mathtt{P})\ \} & \text{if } \varphi_0(\ell) = \emptyset \\ \varphi(\ell) & \text{if } \varphi_0(\ell) \neq \emptyset \end{cases}
\end{aligned}$$

Notice that $\mathit{lift}(\varphi)(\ell) \neq \emptyset$ for all $\ell \in \mathit{labels}(\mathtt{P})$. Second, the definition of $MN(\mathtt{P})$ is:

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

This algorithm is polynomial time: φ_0 takes polynomial time to compute using the technique of Fähndrich and Aiken [4], which intuitively is a fixed-point computation with $\lambda\ell.\emptyset$ as the starting point. Lifting clearly takes polynomial time, and $MN(\mathtt{P})$ 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 $\sqcup MN(\mathtt{P})(\ell)$ equal to $\mathit{static\text{-}type}(\ell, \mathtt{P})$ for any $\ell \in \mathit{labels}(\mathtt{P})$ such that $\varphi_0(\ell) = \emptyset$. Thus the program transformation based on $MN(\mathtt{P})$ will not change the type annotation for the program points labeled by such ℓ.

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

4 Program Transformation

The program transformation is parameterized 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 by the transformation are that:

- it changes some dynamic method invocations to static method invocations,
- 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 because the transformation is restricted to flow analyses that are acceptable and type respecting, so: (i) all flows are nonempty, (ii) nonempty sets of classes admit least upper bounds because FJS is a single-inheritance language, and (iii) the property of being type respecting implies that the new types (that is, the least upper bounds) can only 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.

$$
\begin{array}{ll}
[\![(\overline{\mathtt{CD}},\mathtt{e})]\!]_\varphi & = ([\![\overline{\mathtt{CD}}]\!]_\varphi^{\overline{\mathtt{CD}}},[\![\mathtt{e}]\!]_\varphi^{\overline{\mathtt{CD}}}) \\
[\![\mathtt{class\ C\ extends\ D\ \{}\overline{\mathtt{C}}\ \overline{\mathtt{f}^\ell}\mathtt{;\ K}\ \overline{\mathtt{M}}\mathtt{\}}]\!]_\varphi^{\overline{\mathtt{CD}}} & = \mathtt{class\ C\ extends\ D\ \{} \\
 & \quad \sqcup\varphi(\overline{\ell})\ \overline{\mathtt{f}^\ell}\mathtt{;}\ [\![\mathtt{K}]\!]_\varphi\ \ [\![\overline{\mathtt{M}}]\!]_\varphi^{\overline{\mathtt{CD}}}\mathtt{\}} \\
[\![\mathtt{C(}\overline{\mathtt{D}}\ \overline{\mathtt{f}}\mathtt{)\ \{super(}\overline{\mathtt{f1}}\mathtt{);\ this.}\overline{\mathtt{f2}}\mathtt{=}\overline{\mathtt{f2}}\mathtt{\}}]\!]_\varphi & = \mathtt{C(}\sqcup\varphi(lab(\overline{\mathtt{f}}))\ \overline{\mathtt{f}}\mathtt{)} \\
 & \quad \mathtt{\{super(}\overline{\mathtt{f1}}\mathtt{);\ this.}\overline{\mathtt{f2}}\mathtt{=}\overline{\mathtt{f2}}\mathtt{\}} \\
[\![\mathtt{D\ m(}\overline{\mathtt{E}}\ \overline{\mathtt{x}^\ell}\mathtt{)\ \{return}^\ell\mathtt{e;\}}]\!]_\varphi^{\overline{\mathtt{CD}}} & = \sqcup\varphi(\ell)\ \mathtt{m(}\sqcup\varphi(\overline{\ell})\ \overline{\mathtt{x}^\ell}\mathtt{)\{return}^\ell[\![\mathtt{e}]\!]_\varphi^{\overline{\mathtt{CD}}}\mathtt{;\}} \\
 & \\
[\![\mathtt{x}^\ell]\!]_\varphi^{\overline{\mathtt{CD}}} & = \mathtt{x}^\ell \\
[\![\mathtt{e.f}^\ell]\!]_\varphi^{\overline{\mathtt{CD}}} & = [\![\mathtt{e}]\!]_\varphi^{\overline{\mathtt{CD}}}\mathtt{.f}^\ell \\
[\![\mathtt{new}^\ell\ \mathtt{D(}\overline{\mathtt{e}}\mathtt{)}]\!]_\varphi^{\overline{\mathtt{CD}}} & = \mathtt{new}^\ell\ \mathtt{D(}[\![\overline{\mathtt{e}}]\!]_\varphi^{\overline{\mathtt{CD}}}\mathtt{)} \\
[\![\mathtt{(D)}^\ell\mathtt{e}]\!]_\varphi^{\overline{\mathtt{CD}}} & = (\sqcup\varphi(\ell))^\ell[\![\mathtt{e}]\!]_\varphi^{\overline{\mathtt{CD}}} \\
[\![\mathtt{e.m(}\overline{\mathtt{e}}\mathtt{)}^\ell]\!]_\varphi^{\overline{\mathtt{CD}}} & = [\![\mathtt{e}]\!]_\varphi^{\overline{\mathtt{CD}}}\mathtt{.D::m(}[\![\overline{\mathtt{e}}]\!]_\varphi^{\overline{\mathtt{CD}}}\mathtt{)}^\ell \\
\quad \text{where } \forall \mathtt{E} \in \varphi(lab(\mathtt{e})) : impl(\overline{\mathtt{CD}},\mathtt{E},\mathtt{m}) & = \mathtt{D::m} \\
[\![\mathtt{e.m(}\overline{\mathtt{e}}\mathtt{)}^\ell]\!]_\varphi^{\overline{\mathtt{CD}}} & = [\![\mathtt{e}]\!]_\varphi^{\overline{\mathtt{CD}}}\mathtt{.m(}[\![\overline{\mathtt{e}}]\!]_\varphi^{\overline{\mathtt{CD}}}\mathtt{)}^\ell \\
\quad \text{otherwise} & \\
[\![\mathtt{e.D::m(}\overline{\mathtt{e}}\mathtt{)}^\ell]\!]_\varphi^{\overline{\mathtt{CD}}} & = [\![\mathtt{e}]\!]_\varphi^{\overline{\mathtt{CD}}}\mathtt{.D::m(}[\![\overline{\mathtt{e}}]\!]_\varphi^{\overline{\mathtt{CD}}}\mathtt{)}^\ell \\
 & \\
[\![\mathtt{x}_1 : \mathtt{C}_1, \ldots, \mathtt{x}_n : \mathtt{C}_n]\!]_\varphi & = \mathtt{x}_1 : \sqcup\varphi(lab(\mathtt{x}_1)), \ldots, \mathtt{x}_n : \sqcup\varphi(lab(\mathtt{x}_n))
\end{array}
$$

Fig. 5. The Transformation of Dynamic to Static Dispatch

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

The transformation is also operationally correct, in that the transformed program simulates the original program step for step and vice versa, as stated in 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* $\varphi \in$ *flow-analysis*$(\mathtt{P}_1)$ *then:* $\mathtt{P}_1 \mapsto \mathtt{P}_2$ *if and only if* $[\![\mathtt{P}_1]\!]_\varphi \mapsto [\![\mathtt{P}_2]\!]_\varphi$.

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* $\varphi \in$ *flow-analysis*$(\mathtt{P})$, *then* $\varphi \in$ *flow-analysis*$([\![\mathtt{P}]\!]_\varphi)$.

Given a flow analysis, it is sufficient to apply the transformation only once; applying it again 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* $\varphi \in$ *flow-information*$(\mathtt{P})$ *then* $[\![[\![\mathtt{P}]\!]_\varphi]\!]_\varphi = [\![\mathtt{P}]\!]_\varphi$.

5 Conclusion

Type-safe method inlining can be done for a single-inheritance language without resorting to the insertion of type casts or new types. Our approach is based on flow analysis, and our experience is that it is 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 Class Hierarchy Analysis and our own *MN* analysis satisfy requirements, more work in needed to investigate alternatives. Future work includes extending our approach to the full Java type system which enables multiple subtyping among interfaces.

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 realization that the problem is considerably harder than flow-directed inlining for an untyped language.

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. 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 $\mathit{fields}(\overline{\mathtt{CD}},\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$. Thus, $\mathtt{e}_2$ is a subexpression of $\mathtt{e}_1$. By the conditions for acceptability, $\varphi(\mathit{lab}(\mathtt{e}_i)) \subseteq \varphi(\mathit{lab}(\mathtt{f}_i))$ and $\varphi(\mathit{lab}(\mathtt{f}_i)) = (\ell_2)$. By transitivity, $\varphi(\mathtt{e}_2) = \varphi(\mathit{lab}(\mathtt{e}_i)) \subseteq \varphi(\ell_2) = \varphi(\mathit{lab}(\mathtt{e}_1))$.

cast: In this case $\mathtt{e}_1 = (\mathtt{C})^{\ell_1}\ \mathtt{new}^{\ell_2}\ \mathtt{D}(\overline{\mathtt{e}})$, $\mathtt{e}_2 = \mathtt{new}^{\ell_1}\ \mathtt{D}(\overline{\mathtt{e}})$, and $\overline{\mathtt{CD}} \vdash \mathtt{D} <: \mathtt{C}$. The only subexpressions of $\mathtt{e}_2$ that are not subexpressions of $\mathtt{e}_1$ is $\mathtt{e}_2$ itself. For acceptability, it must be that $\mathtt{D} \in \varphi(\ell_1)$ and $\varphi(\mathit{lab}(\overline{\mathtt{e}})) \subseteq \varphi(\overline{\mathtt{f}})$ where $\mathit{fields}(\overline{\mathtt{CD}},\mathtt{D}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$. The latter follows by the acceptability conditions for $\mathtt{e}_1$. The same conditions also give that $\mathtt{D} \in \varphi(\ell_2)$ and $\varphi(\ell_2) \cap \mathit{subclasses}(\mathtt{P}_1,\mathtt{C}) = \varphi(\ell_1)$. Since $\mathtt{D}$ is a subclass of $\mathtt{C}$, $\mathtt{D} \in \varphi(\ell_1)$ as required. For type respecting, it must be that $\overline{\mathtt{CD}} \vdash \mathtt{E} <: \mathtt{D}$ for each $\mathtt{E} \in \varphi(\ell_1)$ $(*)$. By type respecting for $\mathit{lab}(\mathtt{e}_1)$, $\overline{\mathtt{CD}} \vdash \mathtt{E} <: \mathtt{D}$ for each $\mathtt{E} \in \varphi(\ell_2)$. Since $\varphi(\ell_1) = \varphi(\ell_2) \cap \mathit{subclasses}(\mathtt{P}_1,\mathtt{C})$, $(*)$ holds. Finally, $\varphi(\mathit{lab}(\mathtt{e}_2)) = \varphi(\ell_1) = \varphi(\mathit{lab}(\mathtt{e}_2))$.

dynamic method invocation: In this case $\mathtt{e}_1 = \mathtt{new}^{\ell_1}\ \mathtt{C}(\overline{\mathtt{e}}).\mathtt{m}(\overline{\mathtt{d}})^{\ell_2}$ and $\mathtt{e}_2 = \mathtt{e}\{\mathtt{this},\overline{\mathtt{x}} := \mathtt{new}^{\ell_1}\ \mathtt{C}(\overline{\mathtt{e}}),\overline{\mathtt{d}}\}$ where $\mathit{mbody}(\overline{\mathtt{CD}},\mathtt{C},\mathtt{m}) = (\ell',\overline{\mathtt{x}},\mathtt{e})$. By the conditions for acceptability $\varphi(\mathit{lab}(\overline{\mathtt{d}})) \subseteq \varphi(\mathit{lab}(\overline{\mathtt{x}}))$ and $\varphi(\ell_1) \subseteq \varphi(\mathit{lab}(\mathtt{this}))$. By Lemma 1, φ satisfies the conditions for acceptability and type respecting for $\mathtt{e}_2$ and all its subexpressions. Also by the conditions for acceptability $\varphi(\ell') = \varphi(\ell_2)$ and $\varphi(\mathit{lab}(\mathtt{e})) \subseteq \varphi(\ell')$. Thus $\varphi(\mathit{lab}(\mathtt{e}_2)) = \varphi(\mathit{lab}(\mathtt{e})) \subseteq \varphi(\ell_2) = \varphi(\mathit{lab}(\mathtt{e}_1))$.

static method invocation: In this case $\mathtt{e}_1 = \mathtt{new}^{\ell_1}\ \mathtt{C}(\overline{\mathtt{e}}).\mathtt{D}::\mathtt{m}(\overline{\mathtt{d}})^{\ell_2}$ and $\mathtt{e}_2 = \mathtt{e}\{\mathtt{this},\overline{\mathtt{x}} := \mathtt{new}^{\ell_1}\ \mathtt{C}(\overline{\mathtt{e}}),\overline{\mathtt{d}}\}$ where $\mathit{mbody}(\overline{\mathtt{CD}},\mathtt{D},\mathtt{m}) = (\ell',\overline{\mathtt{x}},\mathtt{e})$. By the conditions for acceptability $\varphi(\mathit{lab}(\overline{\mathtt{d}})) \subseteq \varphi(\mathit{lab}(\overline{\mathtt{x}}))$ and $\varphi(\ell_1) \subseteq \varphi(\mathit{lab}(\mathtt{this}))$. By Lemma 1, φ satisfies the conditions for acceptability and type respecting for $\mathtt{e}_2$ and all its subexpressions. Also by the conditions for acceptability $\varphi(\ell') = \varphi(\ell_2)$ and $\varphi(\mathit{lab}(\mathtt{e})) \subseteq \varphi(\ell')$. Thus $\varphi(\mathit{lab}(\mathtt{e}_2)) = \varphi(\mathit{lab}(\mathtt{e})) \subseteq \varphi(\ell_2) = \varphi(\mathit{lab}(\mathtt{e}_1))$.

Lemma 1. *If φ satisfies the conditions for acceptability and type respecting for all labels in $\mathtt{e}$ and $\overline{\mathtt{d}}$ and if $\varphi(\mathit{lab}(\overline{\mathtt{d}})) \subseteq \varphi(\mathit{lab}(\overline{\mathtt{x}}))$ then φ satisfies the conditions for acceptability and type respecting for all labels in $\mathtt{e}\{\overline{\mathtt{x}} := \overline{\mathtt{d}}\}$.*

Proof. Straightforward.

B Proof of Theorem 4

Theorem 4 follows immediately from Rule (25), and from Lemma 2 and Lemma 4, as stated and proved below.

Lemma 2. *Suppose* $\varphi \in \mathit{acceptable}(\mathtt{P}) \cap \mathit{type\text{-}respecting}(\mathtt{P})$*, 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 Lemma 3 and Lemma 7, using Rule (24).

Lemma 3. *Suppose* $\varphi \in \mathit{acceptable}(\mathtt{P}) \cap \mathit{type\text{-}respecting}(\mathtt{P})$*, 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 4, 5, 8, 9, using Rules (17), (23), (36), (37).

Lemma 4. *Suppose* $\varphi \in \mathit{acceptable}(\mathtt{P}) \cap \mathit{type\text{-}respecting}(\mathtt{P})$*, and* $\mathtt{P} = (\overline{\mathtt{CD}}, \mathtt{e}_0)$*. If* $\overline{\mathtt{CD}}; \Gamma \vdash \mathtt{e} \in \mathtt{D}$*, then* $[\![\overline{\mathtt{CD}}]\!]_\varphi^{\overline{\mathtt{CD}}}; [\![\Gamma]\!]_\varphi \vdash [\![\mathtt{e}]\!]_\varphi^{\overline{\mathtt{CD}}} \in \sqcup\varphi(\mathit{lab}(\mathtt{e}))$*.*

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 to be 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 $\mathit{fields}(\overline{\mathtt{CD}}, \mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}$, where $\mathtt{f}_\mathtt{i}$ occurs in $\overline{\mathtt{f}}$. From the induction hypothesis we have $[\![\overline{\mathtt{CD}}]\!]_\varphi^{\overline{\mathtt{CD}}}; [\![\Gamma]\!]_\varphi \vdash [\![\mathtt{e}_0]\!]_\varphi^{\overline{\mathtt{CD}}} \in \sqcup\varphi(\mathit{lab}(\mathtt{e}_0))$. From $\varphi \in \mathit{type\text{-}respecting}(\mathtt{P})$ we have $\overline{\mathtt{CD}} \vdash \sqcup\varphi(\mathit{lab}(\mathtt{e}_0))$ `<:` $\mathtt{C}_0$ so $\mathit{fields}(\overline{\mathtt{CD}}, \sqcup\varphi(\mathit{lab}(\mathtt{e}_0))) = \overline{\mathtt{D}}\ \overline{\mathtt{g}}\ \overline{\mathtt{C}}\ \overline{\mathtt{f}}$. Hence, from Lemma 7, we have $\mathit{fields}([\![\overline{\mathtt{CD}}]\!]_\varphi^{\overline{\mathtt{CD}}}, \sqcup\varphi(\mathit{lab}(\mathtt{e}_0))) = \sqcup\varphi(\mathit{lab}(\overline{\mathtt{g}}))\ \overline{\mathtt{g}}\ \sqcup\varphi(\mathit{lab}(\overline{\mathtt{f}}))\ \overline{\mathtt{f}}$. From Rule (26) we have $\varphi(\mathit{lab}(\mathtt{f}_\mathtt{i})) = \varphi(\ell)$, so $\sqcup\varphi(\mathit{lab}(\mathtt{f}_\mathtt{i})) = \sqcup\varphi(\ell)$, 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 $\overline{\mathtt{CD}} \vdash \overline{\mathtt{E}}$ `<:` $\overline{\mathtt{D}}$. From Lemma 7 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 9 we have $\overline{\mathtt{CD}} \vdash \sqcup\varphi(\mathit{lab}(\overline{\mathtt{e}}))$ `<:` $\sqcup\varphi(\mathit{lab}(\overline{\mathtt{f}}))$, and so from Lemma 8 we have $[\![\overline{\mathtt{CD}}]\!]_\varphi^{\overline{\mathtt{CD}}} \vdash \sqcup\varphi(\mathit{lab}(\overline{\mathtt{e}}))$ `<:` $\sqcup\varphi(\mathit{lab}(\overline{\mathtt{f}}))$. 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 $\varphi \in \mathit{type\text{-}respecting}(\mathtt{P})$, 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 D in P such that $\forall \mathtt{E} \in \varphi(lab(\mathtt{e}_0)) : impl(\overline{\mathtt{CD}}, \mathtt{E}, \mathtt{m}) = \mathtt{D::m}$. We have

$$\overline{\mathtt{CD}}; \Gamma \vdash \mathtt{e}_0 \in \mathtt{C}_0 \tag{41}$$
$$mtype(\overline{\mathtt{CD}}, \mathtt{C}_0, \mathtt{m}) = \overline{\mathtt{D}} \to \mathtt{C} \tag{42}$$
$$\overline{\mathtt{CD}}; \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{C}} \tag{43}$$

From (41), (43), and the induction hypothesis, we have

$$[\![\overline{\mathtt{CD}}]\!]_\varphi^{\overline{\mathtt{CD}}}; [\![\Gamma]\!]_\varphi \vdash [\![\mathtt{e}_0]\!]_\varphi^{\overline{\mathtt{CD}}} \in \sqcup\varphi(lab(\mathtt{e}_0)) \tag{44}$$
$$[\![\overline{\mathtt{CD}}]\!]_\varphi^{\overline{\mathtt{CD}}}; [\![\Gamma]\!]_\varphi \vdash [\![\overline{\mathtt{e}}]\!]_\varphi^{\overline{\mathtt{CD}}} \in \sqcup\varphi(lab(\overline{\mathtt{e}})) \tag{45}$$

We have $\overline{\mathtt{CD}} \vdash \sqcup\varphi(lab(\mathtt{e}_0))$ <: D, so from Lemma 8, we have

$$[\![\overline{\mathtt{CD}}]\!]_\varphi^{\overline{\mathtt{CD}}} \vdash \sqcup\varphi(lab(\mathtt{e}_0)) \mathrel{\mathtt{<:}} \mathtt{D}, \tag{46}$$

and together with (42), we have $mtype(\overline{\mathtt{CD}}, \mathtt{D}, \mathtt{m}) = \overline{\mathtt{D}} \to \mathtt{C}$. Suppose also $mbody(\overline{\mathtt{CD}}, \mathtt{D}, \mathtt{m}) = (\ell', \overline{\mathtt{x}}, \mathtt{e}')$, From Lemma 6 we have

$$mtype([\![\overline{\mathtt{CD}}]\!]_\varphi^{\overline{\mathtt{CD}}}, \mathtt{D}, \mathtt{m}) = (\sqcup\varphi(lab(\overline{\mathtt{x}}))) \to (\sqcup\varphi(\ell')). \tag{47}$$

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

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

Finally, from Rules (38)–(39) we have

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

so

$$\varphi(lab(\overline{\mathtt{e}})) \subseteq \varphi(lab(\overline{\mathtt{x}})) \tag{48}$$
$$\varphi(\ell') = \varphi(\ell). \tag{49}$$

Thus, from Rule (22), and from (44)–(49), we have that we can derive

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

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, we omit the details.

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

Lemma 5. *Suppose* $\varphi \in acceptable(\texttt{P}) \cap type\text{-}respecting(\texttt{P})$*, and* $\texttt{P} = (\overline{\texttt{CD}},\texttt{e}_0)$*. If* $override(\overline{\texttt{CD}}, \texttt{D}, \texttt{m}, \overline{\texttt{C}} \to \texttt{C}_0)$*,* $\overline{\texttt{CD}} \vdash \texttt{C} <: \texttt{D}$*, and* $mbody(\overline{\texttt{CD}}, \texttt{C}, \texttt{m}) = (\ell, \overline{\texttt{x}}, \texttt{e})$*, then*

$$override([\![\overline{\texttt{CD}}]\!]_\varphi^{\overline{\texttt{CD}}}, \texttt{D}, \texttt{m}, (\sqcup\varphi(lab(\overline{\texttt{x}}))) \to (\sqcup\varphi(\ell))).$$

Proof. Immediate from Lemma 6, using Rule (9), (14)–(16).

Lemma 6. *Suppose* $\varphi \in acceptable(\texttt{P}) \cap type\text{-}respecting(\texttt{P})$*, and* $\texttt{P} = (\overline{\texttt{CD}},\texttt{e}_0)$*. If* $mtype(\overline{\texttt{CD}}, \texttt{D}, \texttt{m}) = \overline{\texttt{D}} \to \texttt{D}_0$*,* $\overline{\texttt{CD}} \vdash \texttt{C} <: \texttt{D}$*, and* $mbody(\overline{\texttt{CD}}, \texttt{C}, \texttt{m}) = (\ell, \overline{\texttt{x}}, \texttt{e})$*, then*

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

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

Lemma 7. *Suppose* $\varphi \in acceptable(\texttt{P}) \cap type\text{-}respecting(\texttt{P})$*, and* $\texttt{P} = (\overline{\texttt{CD}},\texttt{e}_0)$*. If* $fields(\overline{\texttt{CD}}, \texttt{D}) = \overline{\texttt{D}}\ \overline{\texttt{g}}$*, then* $fields([\![\overline{\texttt{CD}}]\!]_\varphi^{\overline{\texttt{CD}}}, \texttt{D}) = \sqcup\varphi(lab(\overline{\texttt{g}}))\ \overline{\texttt{g}}$*.*

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

Lemma 8. *If* $\overline{\texttt{CD}} \vdash \texttt{C} <: \texttt{D}$*, then* $[\![\overline{\texttt{CD}}]\!]_\varphi^{\overline{\texttt{CD}}} \vdash \texttt{C} <: \texttt{D}$*.*

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

Lemma 9. *Suppose* $S, T \in flow(\texttt{P}) \setminus \emptyset$*. If* $S \subseteq T$*, then* $\overline{\texttt{CD}} \vdash \sqcup S <: \sqcup T$*.*

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 $\texttt{P}_1 = (\overline{\texttt{CD}},\texttt{X}\langle\texttt{e}_1\rangle)$ and $\texttt{P}_2 = (\overline{\texttt{CD}},\texttt{X}\langle\texttt{e}_2\rangle)$ where $\texttt{e}_1$ and $\texttt{e}_2$ are given by one of the rules in Figure 2. Clearly $[\![\texttt{P}_i]\!]_\varphi = ([\![\overline{\texttt{CD}}]\!]_\varphi^{\overline{\texttt{CD}}},[\![\texttt{X}]\!]_\varphi^{\overline{\texttt{CD}}}\langle[\![\texttt{e}_i]\!]_\varphi^{\overline{\texttt{CD}}}\rangle)$, so it remains to show that $[\![\texttt{e}_1]\!]_\varphi^{\overline{\texttt{CD}}} \mapsto [\![\texttt{e}_2]\!]_\varphi^{\overline{\texttt{CD}}}$. The interesting cases are when $\texttt{e}_1$ is a cast and when $\texttt{e}_1$ is a dynamic method invocation that is transformed to a static method invocation. Case 1, $\texttt{e}_1 = (\texttt{C})^{\ell_1}\texttt{new}^{\ell_2}\ \texttt{D}(\overline{\texttt{e}})$: In this case $\texttt{e}_2 = \texttt{new}^{\ell_1}\ \texttt{D}(\overline{\texttt{e}})$, $[\![\texttt{e}_1]\!]_\varphi^{\overline{\texttt{CD}}} = (\sqcup\varphi(\ell_1))^{\ell_1}\texttt{new}^{\ell_2}\ \texttt{D}([\![\overline{\texttt{e}}]\!]_\varphi^{\overline{\texttt{CD}}})$, and $\overline{\texttt{CD}} \vdash \texttt{D} <: \texttt{C}$. By the acceptability of φ, $\texttt{D} \in \varphi(\ell_2)$ and $\varphi(\ell_2) \cap subclasses(\texttt{P}_1, \texttt{C}) \subseteq \varphi(\ell_1)$. Thus $\texttt{D} \in \varphi(\ell_1)$, so $\overline{\texttt{CD}} \vdash \texttt{D} <: \sqcup\varphi(\ell_1)$. By the reduction rules $[\![\texttt{e}_1]\!]_\varphi^{\overline{\texttt{CD}}} \mapsto \texttt{new}^{\ell_1}\ \texttt{D}([\![\overline{\texttt{e}}]\!]_\varphi^{\overline{\texttt{CD}}}) = [\![\texttt{e}_2]\!]_\varphi^{\overline{\texttt{CD}}}$. Case 2, $\texttt{e}_1 = \texttt{new}^{\ell_1}\ \texttt{C}(\overline{\texttt{e}}).\texttt{m}(\overline{\texttt{d}})^{\ell_2}$ and $[\![\texttt{e}_1]\!]_\varphi^{\overline{\texttt{CD}}} = \texttt{new}^{\ell_1}\ \texttt{C}([\![\overline{\texttt{e}}]\!]_\varphi^{\overline{\texttt{CD}}}).\texttt{D::m}([\![\overline{\texttt{d}}]\!]_\varphi^{\overline{\texttt{CD}}})^{\ell_2}$: In this case, $\forall \texttt{E} \in \varphi(\ell_1) : impl(\overline{\texttt{CD}}, \texttt{E}, \texttt{m}) = \texttt{D::m}$ and $\texttt{e}_2 = \texttt{e}\{\texttt{this}, \overline{\texttt{x}} := \texttt{new}^{\ell_1}\ \texttt{C}(\overline{\texttt{e}}), \overline{\texttt{d}}\}$ where $mbody(\overline{\texttt{CD}}, \texttt{C}, \texttt{m}) = (\ell, \overline{\texttt{x}}, \texttt{e})$. By the acceptability of φ, $\texttt{C} \in \varphi(\ell_1)$. So $impl(\overline{\texttt{CD}}, \texttt{C}, \texttt{m}) = \texttt{D::m}$. By inspecting the definitions, $mbody(\overline{\texttt{CD}}, \texttt{D}, \texttt{m}) = (\ell, \overline{\texttt{x}}, \texttt{e})$. Thus:

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

($\Leftarrow$) 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 $[\![\mathtt{P}_1]\!]_\varphi \mapsto ([\![\overline{\mathtt{CD}}]\!]^{\overline{\mathtt{CD}}}_\varphi,[\![\mathtt{X}]\!]^{\overline{\mathtt{CD}}}_\varphi\langle\mathtt{e'}\rangle)$ for $[\![\mathtt{e}_1]\!]^{\overline{\mathtt{CD}}}_\varphi$ and $\mathtt{e'}$ as in the rules in Figure 2. It remains to show that $\mathtt{e}_1 \mapsto \mathtt{e}_2$ and $[\![\mathtt{e}_2]\!]^{\overline{\mathtt{CD}}}_\varphi = \mathtt{e}'$. The interesting cases are when $\mathtt{e}_1$ is a cast and when $\mathtt{e}_1$ is a dynamic method invocation that is transformed to a static method invocation. Case 1, $\mathtt{e}_1 = (\mathtt{C})^{\ell_1}\mathtt{new}^{\ell_2}\ \mathtt{D}(\overline{\mathtt{e}})$: In this case $[\![\mathtt{e}_1]\!]^{\overline{\mathtt{CD}}}_\varphi = (\sqcup\varphi(\ell_1))^{\ell_1}\mathtt{new}^{\ell_2}\ \mathtt{D}([\![\overline{\mathtt{e}}]\!]^{\overline{\mathtt{CD}}}_\varphi)$, $\mathtt{e}' = \mathtt{new}^{\ell_1}\ \mathtt{D}([\![\overline{\mathtt{e}}]\!]^{\overline{\mathtt{CD}}}_\varphi)$, and $\overline{\mathtt{CD}} \vdash \mathtt{D}\ \mathtt{<:}\ \sqcup\varphi(\ell_1)$. Since φ is type respecting $\overline{\mathtt{CD}} \vdash \sqcup\varphi(\ell_1)\ \mathtt{<:}\ \mathtt{C}$. By transitivity of subtyping, $\overline{\mathtt{CD}} \vdash \mathtt{D}\ \mathtt{<:}\ \mathtt{C}$. Letting $\mathtt{e}_2$ be $\mathtt{new}^{\ell_1}\ \mathtt{D}(\overline{\mathtt{e}})$ then $\mathtt{e}_1 \mapsto \mathtt{e}_2$ and $[\![\mathtt{e}_2]\!]^{\overline{\mathtt{CD}}}_\varphi = \mathtt{e}'$. Case 2, $\mathtt{e}_1 = \mathtt{new}^{\ell_1}\ \mathtt{C}(\overline{\mathtt{e}}).\mathtt{m}(\overline{\mathtt{d}})^{\ell_2}$ and $[\![\mathtt{e}_1]\!]^{\overline{\mathtt{CD}}}_\varphi = \mathtt{new}^{\ell_1}\ \mathtt{C}([\![\overline{\mathtt{e}}]\!]^{\overline{\mathtt{CD}}}_\varphi).\mathtt{D::m}([\![\overline{\mathtt{d}}]\!]^{\overline{\mathtt{CD}}}_\varphi)^{\ell_2}$: In this case, $\forall \mathtt{E} \in \varphi(\ell_1) : \mathit{impl}(\overline{\mathtt{CD}},\mathtt{E},\mathtt{m}) = \mathtt{D::m}$ and $\mathtt{e}' = [\![\mathtt{e}]\!]^{\overline{\mathtt{CD}}}_\varphi\{\mathtt{this},\overline{\mathtt{x}} := \mathtt{new}^{\ell_1}\ \mathtt{C}([\![\overline{\mathtt{e}}]\!]^{\overline{\mathtt{CD}}}_\varphi),[\![\overline{\mathtt{d}}]\!]^{\overline{\mathtt{CD}}}_\varphi\}$ where $\mathit{mbody}(\overline{\mathtt{CD}},\mathtt{D},\mathtt{m}) = (\ell,\overline{\mathtt{x}},\mathtt{e})$. By the acceptability of φ, $\mathtt{C} \in \varphi(\ell_1)$. So $\mathit{impl}(\overline{\mathtt{CD}},\mathtt{C},\mathtt{m}) = \mathtt{D::m}$. By examination of the auxiliary definitions, $\mathit{mbody}(\overline{\mathtt{CD}},\mathtt{C},\mathtt{m}) = (\ell,\overline{\mathtt{x}},\mathtt{e})$. Letting $\mathtt{e}_2$ be $\mathtt{e}\{\mathtt{this},\overline{\mathtt{x}} := \mathtt{new}^{\ell_1}\ \mathtt{C}(\overline{\mathtt{e}}),\overline{\mathtt{d}}\}$ then $\mathtt{e}_1 \mapsto \mathtt{e}_2$ and $[\![\mathtt{e}_2]\!]^{\overline{\mathtt{CD}}}_\varphi = \mathtt{e}'$.

References

1. Lars Ole Andersen. Self-applicable C program specialization. In *Proceedings of PEPM'92, Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 54–61, June 1992. (Technical Report YALEU/DCS/RR-909, Yale University).
2. Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of European Conference on Object-Oriented Programming*, pages 77–101, Aarhus, Denmark, August 1995. Springer-Verlag (*LNCS* 952).
3. David Detlefs and Ole Agesen. Inlining of virtual methods. In *Proceedings of ECOOP'99, European Conference on Object-Oriented Programming*, pages 258–278. Springer-Verlag (*LNCS* 1628), 1999.
4. Manuel Fähndrich and Alexander Aiken. Program analysis using mixed term and set constraints. In *Proceedings of SAS'97, International Static Analysis Symposium*, pages 114–126. Springer-Verlag (*LNCS*), 1997.
5. Etienne M. Gagnon, Laurie J. Hendren, and Guillaume Marceau. Efficient inference of static types for Java bytecode. In *Proceedings of SAS'00, International Static Analysis Symposium*, pages 199–219. Springer-Verlag (*LNCS* 1824), 2000.
6. Neal Glew. An efficient class and object encoding. In *Proceedings of OOPSLA'00, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 311–324, Minneapolis, Minnesota, October 2000.
7. Nevin Heintze. Control-flow analysis and type systems. In *Proceedings of SAS'95, International Static Analysis Symposium*, pages 189–206. Springer-Verlag (*LNCS* 983), Glasgow, Scotland, September 1995.
8. Atsushi Igarashi, Benjamion Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 132–146, Denver, CO, USA, October 1999.

9. Suresh Jagannathan, Andrew Wright, and Stephen Weeks. Type-directed flow analysis for typed intermediate languages. In *Proceedings of SAS'97, International Static Analysis Symposium*. Springer-Verlag (*LNCS*), 1997.
10. Todd Knoblock and Jakob Rehof. Type elaboration and subtype completion for Java Bytecode. *ACM Transactions on Programming Languages and Systems*, 23(2):243–272, March 2001.
11. Greg Morrisett, David Tarditi, Perry Cheng, Christopher Stone, Robert Harper, and Peter Lee. The TIL/ML compiler: Performance and safety through types. In *ACM SIGPLAN Workshop on Compiler Support for System Software*, Tucson, AZ, USA, February 1996.
12. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Progamming Languages and Systems*, 21(3):528–569, May 1999.
13. Flemming Nielson. The typed lambda-calculus with first-class processes. In *Proceedings of PARLE*, pages 357–373, April 1989.
14. Jens Palsberg. Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems*, 17(1):47–62, January 1995. Preliminary version in Proceedings of CAAP'94, Colloquium on Trees in Algebra and Programming, Springer-Verlag (*LNCS* 787), pages 276–290, Edinburgh, Scotland, April 1994.
15. Jens Palsberg. Equality-based flow analysis versus recursive types. *ACM Transactions on Programming Languages and Systems*, 20(6):1251–1264, 1998.
16. Jens Palsberg and Patrick M. O'Keefe. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems*, 17(4):576–599, July 1995. Preliminary version in Proceedings of POPL'95, 22nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages, pages 367–378, San Francisco, California, January 1995.
17. Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. *Journal of Functional Programming*, 11(3):263–317, May 2001. Preliminary version in Proceedings of POPL'98, 25th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages, pages 197–208, San Diego, California, January 1998.
18. Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1994.
19. Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of POPL'96, 23nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 32–41, 1996.
20. David Tarditi, Greg Morrisett, Perry Cheng, Christopher Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, Philadelphia, PA, USA, May 1996. ACM Press.
21. Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.
22. Andrew Wright, Suresh Jagannathan, Cristian Ungureanu, and Aaron Hertzmann. Compiling Java to a typed lambda-calculus: A preliminary report. In *ACM Workshop on Types in Compilation*, Kyoto, Japan, March 1998.