Transparent Proxies for Java Futures

Polyvios Pratikakis
polyvios@cs.umd.edu

Jaime Spacco
jspacco@cs.umd.edu

Michael Hicks
mwh@cs.umd.edu

Department of Computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

A *proxy* object is a surrogate or placeholder that controls access to another target object. Proxies can be used to support distributed programming, lazy or parallel evaluation, access control, and other simple forms of behavioral reflection. However, *wrapper proxies* (like *futures* or *suspensions* for yet-to-be-computed results) can require significant code changes to be used in statically-typed languages, while proxies more generally can inadvertently violate assumptions of transparency, resulting in subtle bugs.

To solve these problems, we have designed and implemented a simple framework for proxy programming that employs a static analysis based on qualifier inference, but with additional novelties. Code for using wrapper proxies is automatically introduced via a classfile-to-classfile transformation, and potential violations of transparency are signaled to the programmer. We have formalized our analysis and proven it sound. Our framework has a variety of applications, including support for asynchronous method calls returning futures. Experimental results demonstrate the benefits of our framework: programmers are relieved of managing and/or checking proxy usage, analysis times are reasonably fast, overheads introduced by added dynamic checks are negligible, and performance improvements can be significant. For example, changing two lines in a simple RMI-based peer-to-peer application and then using our framework resulted in a large performance gain.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Frameworks,Concurrent programming structures,Control structures*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Object-oriented Programming*

General Terms

Languages, Experimentation, Reliability, Theory

OOPSLA 'O4 Oct. 24-28, 2004, Vancouver, British Columbia, Canada.

Keywords

Java, future, proxy, type inference, type qualifier

1. INTRODUCTION

A *proxy* object is a surrogate or placeholder that controls access to another object. One example of a proxy is a *future*, popularized in MultiLisp [23]. In MultiLisp, the syntax `(future` e`)` designates that expression e should be evaluated concurrently. A future for it is returned, and some time later the program *claims* the future, possibly blocking until the result of evaluating e is available. For example, in the following code, the two lists `x` and `y` are sorted in parallel, the former in a new thread, and the latter in the parent thread:

```
(merge (future (mergesort x)) (mergesort y))
```

The results of both `mergesort` computations are passed to the `merge` routine; the first argument will be a future while the second argument will be a sorted list.

In MultiLisp, claims are performed transparently by the interpreter. In our example, this allows the programmer to write `merge` as if it takes two sorted lists as arguments, and the interpreter will perform claims as necessary. In general, the programmer simply inserts `future` annotations in the program and the runtime transparently takes care of the rest.[1] This makes the use of futures simple and lightweight.

A future is an example of a *wrapper proxy* in that it wraps the actual result; whenever the actual result is needed, the future must be unwrapped to retrieve it. Other examples of wrapper proxies include *suspensions*, which are wrappers for lazy computations, and *capabilities*, which are wrappers for controlled resources.

We would like to support wrapper proxies in Java with the same kind of transparency afforded by MultiLisp. To add futures, we would provide *asynchronous* method calls to return a future for a non-`void` result. Existing proposals to do this [28, 24, 35] fall short of our goal because they make futures manifest to the programmer. For example, Java 1.5's `util.concurrent` library [24] defines a future with the following Java interface:

```
public interface Future<V> {
  V get();
  V get(long timeout, TimeUnit unit);
  ...
}
```

Introducing `util.concurrent` futures into a Java program thus imposes two programming tasks. First, whenever a `Future<`V`>` value could be passed to a function, the function's type must be changed. In our example, we would have

[1]There is the possible need for added synchronization due to side-effects in futurized computations.

to change the type of `merge` to take a `Future<List>` as its first argument (or `Object` if `merge` could also be called with normal `List` objects).

Second, all futures must be claimed manually by calling `get`. For example, the `merge` function would claim the `Future<List>` to store its values in the merged list. We might also claim a future o to avoid revealing its identity, e.g., in the expression o==o'. Not doing so could lead to subtle bugs which we call *transparency violations*. For example, when o is the future wrapping o', then o==o' would be false, which could result in unexpected behavior, such as storing a future and its value in the same container. Changing types and adding claims can require considerable programming effort, whether to add futures or to later remove them.

To solve these problems, we have developed a framework for proxy programming. At the core of the framework is a static analysis that tracks how a proxy might flow through the program, coupled with a transformation to implement proxy manipulations at runtime. To customize the framework, a programmer specifies the syntactic points where a proxy is introduced (e.g., by specifying a method call is asynchronous), and the expression forms that require a claim (e.g., when a proxy is an argument to `==`). The programmer also provides the code that implements the claim. We have used our framework for a variety of applications:

- We have implemented support for transparent futures. The programmer indicates when a method call should be asynchronous, and specifies a *thread manager* for handling the call. Thread managers include global thread pools, per-object thread pools, and others. Programmers can also influence where futures are claimed. In essence, the framework drastically simplifies programming with Futures in `util.concurrent`, which is timely given the recent release of Java 1.5.

- We have implemented support for transparent suspensions. The programmer annotates when a method call should be performed lazily, and the call is delayed until its suspension is claimed.

- We have implemented an analysis to discover possible transparency violations due to the introduction of *interface proxies* in large programs. An interface proxy shares an interface with its target object, as specified by the proxy design pattern [18]. As with wrapper proxies, incorrect usage of these proxies could result in transparency violations.

Our static analysis is based on *qualifier inference* [15], but improves on it in two ways. First, we support dynamic coercions, needed to claim futures and other wrapper proxies. Second, we use a simple form of flow-sensitivity to avoid claiming the same expression more than once. While our framework was developed for proxy programming, these advances apply to qualifier systems in general. As described in Section 3.8, they enable a number of new or improved applications, including tracking security-sensitive data in a program [37], and supporting stack allocation and non-null types [13].

1.1 Contributions

This paper describes the design, theory, implementation, and evaluation of a framework for proxy programming. We make the following contributions:

- We formalize the problem of transparent proxy programming as one of qualifier inference, extending existing algorithms to support dynamic coercions and a form of flow-sensitivity. We have formalized our analysis as an extension of Featherweight Java (FJ) [21], and proven it sound (Section 3). We are the first to consider qualifier inference in an object-oriented setting, and our approach enables new or improved applications of qualifier systems (Section 3.8).

- We present the design and implementation of three applications of our framework (described above): programming with transparent futures and suspensions (Section 4), and discovering transparency violations (Section 5.4).

- We evaluate the framework's performance on our three applications (Section 5). Analysis times are comparable to those of similar static analyses, and overheads due to inserted claims are negligible. Section 5.3 describes how we profitably used futures and suspensions together in an RMI-based peer-to-peer application: changing two lines resulted in a large performance gain. Section 5.4 describes how our transparency analysis discovered a number of potential transparency violations arising from the introduction of interface proxies in large programs.

2. OVERVIEW

In this section, we present an overview of our framework, including the API seen by the user, and the basic flavor of our static analysis.

2.1 User API

As inputs, our framework takes application and library classfiles to analyze, and a proxy *policy* and *implementation* specification (a *pspec* and *ispec*, respectively). As outputs, the framework produces modified application and library classfiles which form the new application. The *pspec* and *ispec* allow the user to customize the framework to support different kinds of proxies. In particular, the *pspec* defines syntactic patterns in the program that indicate where proxies should be introduced and coerced, while the *ispec* indicates how proxy introduction and coercion are implemented at runtime.

The framework itself consists of two parts: a static analysis (which uses the *pspec*) and a program transformation (which uses the *ispec*). The static analysis discovers where proxies are introduced in the program and then tracks their flow. The analysis observes when a proxy could flow to a location requiring a non-proxy, thus requiring a *coercion* to convert the proxy to a non-proxy. Based on the results of static analysis, the program transformation generates a modified program. In particular, the code at each proxy introduction site is modified to actually create the proxy at runtime, and code is inserted at each coercion site to implement the proxy-to-non-proxy coercion.

As an example, consider how we implement asynchronous method calls in Java using this API (more details are in Section 4). The proxy *pspec* and *ispec* are as follows:

Policy Spec Proxies are introduced by method calls marked by the user as being asynchronous. All expressions that are identity-revealing, e.g., dynamic downcasts or

subexpressions of `instanceof`, must operate on non-proxies (thus necessitating a possible coercion). Moreover, any concrete usage of an object, such as invocations of its methods or extractions of its fields, requires that it be a non-proxy.

Implementation Spec Calls marked as asynchronous are replaced by code that (1) executes the original call in a separate thread, and (2) returns a `Future` as a placeholder for the eventual result. Coercing a possible future requires checking that it is indeed a `Future` (the analysis may have been imprecise), and if so, calling its `get` method to extract the underlying object. This may entail waiting until the result is available.

Lazy method calls are supported similarly, and other applications are described in Section 3.8 and 5.4. Further implementation details are presented in Section 4.1.

Our goal is for the framework to be used during normal software development: the programmer develops the annotated files, and the framework generates the final bytecode. Alternatively, the framework could be used to add needed features to a Java program; the annotated files would simply direct the transformation, and development would proceed with the modified files. This would allow programmers to manually optimize the compiled code, but would eliminate the benefits of the lighter-weight, specification-based use of proxies during development.

We now turn to an overview of our analysis.

2.2 Proxies as Qualifiers

Conceptually, whether or not a particular program variable refers to a proxy is independent of that variable's type. As such, we can think about proxies using *type qualifiers*, which refine the meaning of a particular type. A qualified type is written $Q\ \tau$, where Q is a qualifier and τ is a type. A familiar use of a type qualifier is final: a variable with this qualifier must be immutable, whatever the variable's actual type may be. Proxies can be annotated in the same way. A variable with qualifier nonproxy is definitely *not* a proxy, while one with qualifier proxy may or may not be a proxy. Qualified types admit a natural subtyping relationship. In particular, $\mathsf{nonproxy}\ \tau \leq \mathsf{proxy}\ \tau$. That is, a τ object that is definitely not a proxy can be used where a τ that may or may not be a proxy is expected.

The problem solved by our framework is akin to *qualifier inference* [15]. When using qualifier inference, the programmer annotates expressions that introduce values with a particular qualified type. The inference algorithm determines how these values flow through the program to ensure they are used correctly. Existing qualifier inference systems are not sufficient to model wrapper proxies like futures because they treat qualifiers as having no runtime effect. Creating a future requires spawning a thread and creating a placeholder for its result. Moreover, using a wrapper proxy in a context expecting a nonproxy should not signal an error, but rather should induce a runtime *claim* to acquire the underlying result.

Our analysis augments qualifier inference to support *coercions*. In particular, our formal target language (Section 3) includes an expression form **coerce** e, whose type is the same as that of e but has qualifier nonproxy. During qualifier inference, expression forms in the user's *pspec* drive where coercions are inserted. At runtime, the coercions are implemented following the user's *ispec*. For example, for a possible wrapper proxy e, a dynamic coercion is inserted to convert $e.m()$ to be $(\textbf{coerce}\ e).m()$. At runtime, this coercion is implemented by checking whether e is indeed a proxy, and if so extracting its underlying object to call method m. As an optimization, if e is a local variable `x`, then `x` is treated flow-sensitively by the analysis: the type of `x` following a coercion will have qualifier nonproxy. To justify this flow-sensitivity, code for a coercion logically assigns the coerced value back to source variable `x`.

We can easily generalize our support for flow-sensitive coercions to apply it to traditional qualifier systems. This leads to new or improved applications, as described in Section 3.8.

3. FORMAL DEVELOPMENT

This section describes our analysis formally and proves it sound. We model the analysis as an extension to Featherweight Java (FJ) [21], a purely-functional object calculus. We define an implicitly-typed calculus, which we call FJ^i_Q, and an explicitly-typed calculus, called FJ_Q. Source programs are written in FJ^i_Q, and these are translated into programs in FJ_Q, making manifest operations for manipulating proxies. This translation occurs in two stages, inference and transformation, formalized as follows:

- The judgment $\Gamma \vdash_i e : T; \Gamma'$ defines proxy inference for an expression e in the language FJ^i_Q. A derivation induces two sets of subtyping constraints $\mathcal{F}$ and $\mathcal{C}$. The $\mathcal{F}$ constraints capture how proxies flow through the program, and the $\mathcal{C}$ constraints indicate where coercions could be inserted. The judgment states that, assuming the generated constraints have a solution, expression e has type T in context Γ. Flow-sensitivity is modeled with *output context* Γ', which has the same domain as Γ, but for which some variables may have nonproxy qualifiers rather than proxy qualifiers, as a result of evaluating expression e.Constraints are solved using standard techniques.

- The judgment $\mathcal{T}[\![e]\!] \Rightarrow e$ defines the transformation of the original implicitly-typed FJ^i_Q program into an explicitly-typed program in the language FJ_Q. The $\mathcal{T}[\![\cdot]\!]$ function uses the solutions to the constraints to add coercions where needed, and to fill in needed qualifier and type annotations. The resulting FJ_Q expression e can be typechecked in an explicitly-typed system, described by the judgment $\Gamma \vdash e : T; \Gamma'$. We can show that our system is *sound*: those FJ^i_Q programs for which inference is successful will always type-check, which in turn implies that they will not "go wrong" during execution. We establish this result by defining an operational semantics for FJ_Q and proving standard type soundness and inference soundness theorems.

We present the syntax of the implicitly-typed language FJ^i_Q, define the process of inference and transformation described above, and conclude with the relevant soundness theorems. Additional details can be found in the Appendix.

3.1 Syntax

The syntax of the implicitly-typed calculus FJ^i_Q is shown in Figure 1. Expressions e consist of a "raw" expression

Terms:

$$
\begin{array}{rcl}
CL & ::= & \textbf{class } C \textbf{ extends } C \ \{ \ \bar{T} \ \bar{f}; K \ \bar{M} \ \} \\
K & ::= & C(\bar{T} \ \bar{f}) \ \{ \ \textbf{super}(\bar{f}); \ \textbf{this}.\bar{f} = \bar{f}; \ \} \\
M & ::= & T \ m(\bar{T} \ \bar{x}) \ \{ \ \textbf{return } e; \} \\
\mathcal{E} & ::= & x \mid e.f \mid e.m(\bar{e}) \mid \textbf{new } C(\bar{e}) \mid (C)e \\
 & \mid & \textbf{let } x = e \textbf{ in } e \mid \textbf{makeproxy } e \\
 & \mid & \textbf{if } e = e \textbf{ then } e \textbf{ else } e \\
e & ::= & \mathcal{E}^l
\end{array}
$$

Types:

$$
\begin{array}{rcl}
C, D, E & & \text{class names} \\
Q & ::= & \mathsf{proxy} \mid \mathsf{nonproxy} \mid \kappa \\
\varphi & ::= & \{C_1, \ldots C_n\} \mid \alpha \\
N & ::= & \varphi^C \\
S, T, U & ::= & Q \ N
\end{array}
$$

Figure 1: Syntax of FJ^i_Q

$$\text{SubRef}\frac{}{C \leq C}$$

$$\text{SubTrans}\frac{C \leq D \quad D \leq E}{C \leq E}$$

$$\text{SubDef}\frac{CT(C) = \textbf{class } C \textbf{ extends } D \ \{ \ldots; \ldots \}}{C \leq D}$$

$$\text{SubN}\frac{\{C_1, \ldots C_n\} \subseteq \{D_1, \ldots D_n\} \quad C_0 \leq D_0 \quad D_i \leq D_0 \quad C_i \leq C_0 \quad \text{for all } i > 0}{\{C_1, \ldots C_n\}^{C_0} \leq \{D_1, \ldots D_n\}^{D_0}}$$

$$\text{SubQConst}\frac{}{\mathsf{nonproxy} \leq \mathsf{proxy}}$$

$$\text{SubTyp}\frac{Q \leq Q' \quad N \leq N'}{Q \ N \leq Q' \ N'}$$

Figure 2: FJ_Q and FJ^i_Q: Subtyping

$\mathcal{E}$ and a unique label l, used to designate where coercions should be inserted following inference. There is no explicit coercion expression; these are only present in the target language FJ_Q.

As in FJ, programs consist of a *class table* CT, which maps class names to class definitions CL. Each class definition defines a list of fields $\bar{T} \ \bar{f}$, a constructor K, and a list of methods $\bar{M}$. Constructors K merely assign their arguments to fields, either directly or by invoking the superclass constructor. Method bodies consist of a single expression e. We write $\bar{x}$ as shorthand for $x_1, \ldots, x_n$ (similarly for $\bar{C}$,$\bar{f}$, etc.) and write $\bar{M}$ for $M_1 \ldots M_n$ (no commas). We abbreviate operations on pairs of sequences similarly, writing $\bar{T} \ \bar{f}$ for $T_1 \ f_1, \ldots, T_n \ f_n$, where n is the length of $\bar{T}$ and $\bar{f}$. Sequences of field declarations, parameter names, and method declarations are assumed to contain no duplicate names. Note that **this** is not syntactically different than any other variable, but we typeset it in bold for emphasis, and similarly for **Object**.

Most expressions e are as in FJ, including field access $e.f$, method invocation $e.m(\bar{e})$, object creation **new** $C(\bar{e})$, and cast $(C)e$. We also have support for local variables (**let**s) and **if then else** expressions to illustrate effects of flow sensitivity, described below. Programmers use the expression **makeproxy** e to designate or create a proxy. Our formalism treats proxies generically, ignoring how particular proxies might be implemented. In particular, the operational semantics merely "tags" the result of evaluating e as being a possible proxy.

Types T consist of a qualifier Q and a *set type* N. Set types are a set φ of class names $\{C_1, \ldots, C_n\}$ coupled with a *upper bound* C which must be a supertype of all the C_i. Set types are a technical device to allow inference to be more precise; we do not expect programmers to use them directly. In essence, the set type's upper bound is what one would write in a normal Java program, and the set provides a more precise refinement (which will be determined by inference). For example, say we have defined classes A, B, and C, where B and C are subclasses of A. If some variable `x` could be assigned objects of either class B or C, in a normal Java program we would give `x` type A. In FJ^i_Q, we can give `x` type $\{B, C\}^A$, indicating that `x` will only ever be assigned objects of classes B and/or C, but not objects of type A. Note that checked casts refer to class names C, rather than types T—no qualifier is necessary because it is assumed to be nonproxy, and no set type is necessary as the inference system will infer it.

Proxy inference takes a normal Java program and infers the necessary qualifiers, set-types, and coercions. We model this in FJ^i_Q by extending qualifiers Q with variables κ, and sets of class names φ with variables α. These stand for as-yet-unknown qualifiers and sets of class names, which will be solved for during inference. In the simplest case, we could automatically decorate a normal Java program with fresh variables before performing inference. For example, a Java variable declaration C `x` would be rewritten to be $\kappa \ \alpha^C$ `x`, for fresh κ and α. In fact, the inference rules require explicit types to have this form. In our implementation, we allow users to decorate Java types with qualifiers manually, to implement coercion policies. For example, if a user wished to ensure that no proxies are stored in the `Set` class, she could decorate all relevant `Set` methods to require that input arguments have qualifier nonproxy.

As with FJ, FJ^i_Q does not support mutation (although the flow-sensitivity of coercions updates local variables' types implicitly): all objects are purely functional. This avoids unnecessary complication in the formalism, though our implementation handles the full Java language. Further discussion can be found in Section 3.7.

3.2 Subtyping

Rules for subtyping are shown in Figure 2. These are FJ's subtyping rules extended to consider set types N and qualified types T. The (SubN) rule indicates that a set type N is a subtype of M if N's bound is a subtype of M's, and if N's set is a subset of M's. We include a well-formedness condition here for convenience, stating that all of the types in N's set must be subtypes of N's bound. Subtyping between qualified types using the (SubTyp) rule is natural. For example, if B and C are subclasses of A, given that nonproxy $\leq$ proxy then nonproxy $\{B\}^B \leq$ proxy $\{B, C\}^A$. That is, an object that is definitely not a proxy of class B can be used where a possible proxy of either class B or C, both subtypes of A, is expected.

$$\text{Fields-Object}\ \frac{}{\mathit{fields}(\mathbf{Object}) = \cdot}$$

$$\text{Fields-C}\ \frac{CT(C) = \mathbf{class}\ C\ \mathbf{extends}\ D\ \{\ \bar{T}\ \bar{f}; K\ \bar{M}\ \} \quad \mathit{fields}(D) = \bar{U}\ \bar{g}}{\mathit{fields}(C) = \bar{U}\ \bar{g}, \bar{T}\ \bar{f}}$$

$$\text{Fields-N}\ \frac{\mathit{fields}(C) = \bar{T}\ \bar{f}}{\mathit{fields}(\varphi^C) = \bar{T}\ \bar{f}}$$

$$\text{MType-C}\ \frac{CT(C) = \mathbf{class}\ C\ \mathbf{extends}\ D\ \{\ \bar{T}\ \bar{f}; K\ \bar{M}\ \} \quad U\ m(\bar{T}\ \bar{x})\ \{\ \mathbf{return}\ e;\ \} \in \bar{M}}{\mathit{mtype}(m, C) = \bar{T} \to U}$$

$$\text{MType-CSub}\ \frac{CT(C) = \mathbf{class}\ C\ \mathbf{extends}\ D\ \{\ \bar{T}\ \bar{f}; K\ \bar{M}\ \} \quad m \text{ not defined in } \bar{M}}{\mathit{mtype}(m, C) = \mathit{mtype}(m, D)}$$

$$\text{MType-N}\ \frac{\mathit{mtype}(m, C_i) = \bar{T}_i \to U_i \quad \text{for all } i}{\mathit{mtype}(m, \{C_1, \ldots C_n\}^{C_0}) = \bar{T}_1 \to U_1, \ldots \bar{T}_n \to U_n}$$

$$\text{MBody-C}\ \frac{CT(C) = \mathbf{class}\ C\ \mathbf{extends}\ D\ \{\ \bar{T}\ \bar{f}; K\ \bar{M}\ \} \quad U\ m(\bar{T}\ \bar{x})\ \{\ \mathbf{return}\ e;\ \} \in \bar{M}}{\mathit{mbody}(m, C) = (\bar{x}, e)}$$

$$\text{MBody-CSub}\ \frac{CT(C) = \mathbf{class}\ C\ \mathbf{extends}\ D\ \{\ \bar{T}\ \bar{f}; K\ \bar{M}\ \} \quad m \text{ not defined in } \bar{M}}{\mathit{mbody}(m, C) = \mathit{mbody}(m, D)}$$

$$\text{Override}\ \frac{\begin{array}{c}\mathit{mtype}(m, D) = (\kappa_1\ \varphi_1^{D_1}, \ldots, \kappa_n\ \varphi_n^{D_n}) \to \kappa_0\ \varphi_0^{D_0} \\ \bar{T} \to U = (\kappa_{n+2}\ \varphi_{n+2}^{C_1}, \ldots, \kappa_{2n+1}\ \varphi_{2n+1}^{C_n}) \to \kappa_{n+1}\ \varphi_{n+1}^{C_0} \\ \bar{C} = \bar{D} \qquad C_0 = D_0\end{array}}{\mathit{override}(m, D, \bar{T} \to U)}$$

$$\text{Call}\ \frac{\begin{array}{c}\text{for each } C_i \le C \text{ where } \mathit{mtype}(m, C_i) = \bar{T}_i \to Q_i\ \varphi_i^D \\ C_i \in \varphi \Rightarrow (\bar{S} \le \bar{T}_i\ \&\ Q_i\ \varphi_i^D \le \kappa\ \alpha^D) \\ \kappa, \alpha \text{ fresh}\end{array}}{\mathit{call}(m, \varphi^C, \bar{S}) = \kappa\ \alpha^C}$$

Figure 3: FJ_Q and FJ_Q^i: Auxiliary Definitions

$$\text{I-Var}\ \frac{}{\Gamma[x \mapsto T] \vdash_i x^l : T; \Gamma[x \mapsto T]}$$

$$\text{I-Let}\ \frac{\Gamma \vdash_i e_1 : T; \Gamma_1 \quad \Gamma_1[x \mapsto T] \vdash_i e_2 : T'; \Gamma'[x \mapsto T'']}{\Gamma \vdash_i (\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2)^l : T'; \Gamma'}$$

$$\text{I-If}\ \frac{\begin{array}{c}\Gamma \vdash_c e_1 : \mathsf{nonproxy}\ N_1; \Gamma_1 \\ \Gamma_1 \vdash_c e_2 : \mathsf{nonproxy}\ N_2; \Gamma_2 \\ \Gamma_2 \vdash_i e_3 : Q_3\ \varphi_3^C; \Gamma_3 \quad \Gamma_3 \vdash_i e_4 : Q_4\ \varphi_4^D; \Gamma_4 \\ T' = \kappa\ \alpha^E \qquad Q_3\ \varphi_3^C \le T' \qquad Q_4\ \varphi_4^D \le T' \\ E = \mathit{lub}(C, D) \qquad \Gamma' = \mathit{merge}(\Gamma_3, \Gamma_4)\end{array}}{\Gamma \vdash_i (\mathbf{if}\ e_1 = e_2\ \mathbf{then}\ e_3\ \mathbf{else}\ e_4)^l : T'; \Gamma'}$$

$$\text{I-Field}\ \frac{\Gamma \vdash_c e : \mathsf{nonproxy}\ N; \Gamma' \quad \mathit{fields}(N) = \bar{T}\ \bar{f}}{\Gamma \vdash_i (e.f_i)^l : T_i; \Gamma'}$$

$$\text{I-Invoke}\ \frac{\Gamma \vdash_c e_0 : \mathsf{nonproxy}\ \varphi^C; \Gamma' \quad \Gamma' \vdash_i \bar{e} : \bar{S}; \Gamma'' \quad \mathit{call}(m, \varphi^C, \bar{S}) = \kappa\ \alpha^C}{\Gamma \vdash_i (e_0.m(\bar{e}))^l : \kappa\ \alpha^C; \Gamma''}$$

$$\text{I-New}\ \frac{\mathit{fields}(\{C\}^C) = \bar{T}\ \bar{f} \quad \Gamma \vdash_i \bar{e} : \bar{S}; \Gamma' \quad \bar{S} \le \bar{T}}{\Gamma \vdash_i (\mathbf{new}\ C(\bar{e}))^l : \mathsf{nonproxy}\ \{C\}^C; \Gamma'}$$

$$\text{I-Cast}\ \frac{\Gamma \vdash_c e : \mathsf{nonproxy}\ \varphi^D; \Gamma' \quad \alpha = \mathit{subtypes}(C) \cap \varphi \quad \alpha \text{ fresh}}{\Gamma \vdash_i ((C)e)^l : \mathsf{nonproxy}\ \alpha^C; \Gamma'}$$

$$\text{I-MakeProxy}\ \frac{\Gamma \vdash_c e : \mathsf{nonproxy}\ N; \Gamma'}{\Gamma \vdash_i (\mathbf{makeproxy}\ e)^l : \mathsf{proxy}\ N; \Gamma'}$$

$$\text{I-CoerceExp}\ \frac{\Gamma \vdash_i \mathcal{E}^{l_0} : Q\ N; \Gamma' \quad \mathcal{E} \neq x \quad l \text{ fresh} \quad \mathsf{proxy} \le Q \Rightarrow l \in L}{\Gamma \vdash_c \mathcal{E}^l : \mathsf{nonproxy}\ N; \Gamma'}$$

$$\text{I-CoerceVar}\ \frac{\Gamma \vdash_i x^{l_0} : Q\ N; \Gamma \quad l \text{ fresh} \quad \Gamma = \Gamma'[x \mapsto Q\ N] \quad \mathsf{proxy} \le Q \Rightarrow l \in L}{\Gamma \vdash_c x^l : \mathsf{nonproxy}\ N; \Gamma'[x \mapsto \mathsf{nonproxy}\ N]}$$

Figure 4: FJ_Q^i: Inference for Expressions

$$\text{I-Method}\ \frac{\begin{array}{c}\bar{x} : \bar{T}, \mathbf{this} : \mathsf{nonproxy}\ \{C\}^C \vdash_i e : U; \Gamma' \quad U \le S \\ CT(C) = \mathbf{class}\ C\ \mathbf{extends}\ D\ \{\ \ldots; \ldots\ \} \\ \mathit{override}(m, D, \bar{T} \to S) \\ S = \kappa\ \alpha^C \qquad \bar{T} = \kappa_1\ \alpha_1^{C_1}, \ldots \kappa_n\ \alpha_n^{C_n} \\ \kappa, \kappa_i, \alpha, \alpha_i \text{ fresh}\end{array}}{\vdash_i S\ m(\bar{T}\ \bar{x})\ \{\ \mathbf{return}\ e;\}}$$

$$\text{I-Class}\ \frac{\begin{array}{c}K = C(\bar{T}\ \bar{g}, \bar{S}\ \bar{f})\ \{\ \mathbf{super}(\bar{g});\ \mathbf{this}.\bar{f} = \bar{f};\ \} \\ \mathit{fields}(D) = \bar{T}\ \bar{g} \qquad \bar{T} = \kappa_1\ \alpha_1^{C_1}, \ldots \kappa_n\ \alpha_n^{C_n} \\ \bar{S} = \kappa_1'\ \alpha_1'^{C_1}, \ldots \kappa_n'\ \alpha_n'^{C_n} \qquad \kappa_i, \kappa_i', \alpha_i, \alpha_i' \text{ fresh} \\ \vdash_i \bar{M}\end{array}}{\vdash_i \mathbf{class}\ C\ \mathbf{extends}\ D\ \{\ \bar{T}\ \bar{f}; K\ \bar{M}\ \}}$$

Figure 5: FJ_Q^i: Inference for Classes and Methods

3.3 Inference

Inference is expressed as the judgment $\vdash_i CL$ for class definitions, $\vdash_i M$ for method definitions, and $\Gamma \vdash_i e : T; \Gamma'$ for expressions. The rules are in Figures 4 and 5. The judgment $\Gamma \vdash_i e : T; \Gamma'$ indicates that in context Γ, expression e has type T and output context Γ'.

The rules specify that a nonproxy is required by appealing to the *coercion judgment* $\Gamma \vdash_c e : T; \Gamma'$ (notice the subscript c on $\vdash_c$ rather than i). For example, the (I-Field) rule, which checks an expression $(e.f_i)^l$, indicates that the receiver e must be a nonproxy by including the requirement $\Gamma \vdash_c e : \mathsf{nonproxy}\ N; \Gamma'$ in the premise. In our implementation, those expressions that require nonproxy are determined by the user's *pspec*. For simplicity, the rules presented in Figure 4 are specialized for the case of wrapper proxies. In this case, a nonproxy type implies that operations must occur on the underlying object, rather than on a wrapper proxy.

The coercion judgment is used to note the labels of expressions that may need an inserted coercion. It has two forms. The (I-CoerceExp) rule creates an *implication constraint* that if the qualifier of the given expression e is not nonproxy, then a fresh label l for e is included in a set L. (The fact that this label is fresh simplifies the proof, but is not otherwise important.) This set is used during the trans-

formation to determine where coercions must be inserted. The output type of this judgment always has a nonproxy qualifier; this will be justified by inserting coercions during transformation. The (I-CoerceVar) rule is similar, except that the variable x in the input context is re-bound in the output context to its coerced type. This flow-sensitive treatment allows the continuation avoid coercing a variable that has already been coerced.

Most inference rules thread the output context of one subexpression to the input context of another. When typing $\Gamma \vdash \bar{e} : \bar{T}; \Gamma'$, the output context Γ_i from typing expression e_i is used as the input context when typing e_{i+1}.

Here are highlights of the other interesting rules:

- In the (I-Let) rule, the output context Γ_1 of the binding expression e_1 is extended with the binding $x \mapsto T$ when used as the input context of the body e_2. When typechecking of the body is completed, the x binding is removed from output context Γ'.
- In the (I-If) rule, the output context is a merging of the output context of each of the branches of the **if**. In particular, the function $merge(\Gamma_1, \Gamma_2)$ is the context Γ' such that for each x in $dom(\Gamma_1) \cap dom(\Gamma_2)$, $\Gamma'(x) = T$ where $\Gamma_1(x) \leq T$ and $\Gamma_2(x) \leq T$. The result type is T', which is a supertype of the types of each of the **if** branches, bounded by the least of upper bound of their bounds.
- The (I-Invoke) rule creates subtyping constraints between the arguments $\bar{S}$ and all methods that are possible receivers of the call using the auxiliary function $call(m, \varphi^C, \bar{S})$ (this and other auxiliary functions are shown in Figure 3). This is done using implication constraints: for all possible subtypes of C, only those that appear in φ are constrained. This allows overriding methods to have arguments with different qualifiers than the methods they are overriding, improving the precision of the analysis. For example, the argument o to class A's method m might by a nonproxy, while the argument to its subclass B's overriding method m could be a proxy. This is sound because all calling contexts of m are considered.
- The (I-Cast) rule requires that the resulting set-type α contains those names in e's set-type φ, limited to those that are also subtypes of the bound C. The predicate $subtypes(C)$ is the set of all subtypes of C defined in the class table CT. There are three possible outcomes. First, if C is a subtype of D, then φ may contain classes B such that $C \leq B$. These will be pruned from the solution, since this is a downcast. Second, if C is a supertype of D, then the intersection will be φ, since all the class names in φ, which are bounded by D, are also bounded by C. Finally, if neither situation holds, which is to say that C and D are unrelated, then the intersection will be empty, signaling that we have a type error.
- The (I-MakeProxy) rule requires that e be a nonproxy in **makeproxy** e. This prevents proxies of proxies. While not technically necessary, it simplifies our implementation of coercions. For example, for a wrapper proxy, the underlying object can always be extracted directly; otherwise a coercion would have to iterate until it reached a non-wrapper.

$$
\begin{array}{lll}
\mathcal{F} & \cup & \{Q\ \varphi^C \leq Q'\ \varphi'^D\} \equiv \\
\mathcal{F} & \cup & \{Q \leq Q'\} \cup \{\varphi \subseteq \varphi'\} \cup \\
 & & \{\varphi \subseteq subtypes(C)\} \cup \{\varphi' \subseteq subtypes(D)\} \cup \\
 & & \{C \leq D\} \\
 & & \\
\mathcal{F} & \cup & \{\{D\} \subseteq \alpha\} \cup \{D \in \alpha^C \Rightarrow (\bar{S} \leq \bar{T}_i\ \&\ Q_i\ \varphi_i^C \leq \kappa\ \alpha^C)\} \equiv \\
\mathcal{F} & \cup & \{\{D\} \subseteq \alpha\} \cup \{\bar{S} \leq \bar{T}_i\} \cup \{Q_i\ \varphi_i^C \leq \kappa\ \alpha^C\} \\
 & & \\
\mathcal{F} & \cup & \{\alpha \subseteq (subtypes(C) \cap \varphi)\} \equiv \\
\mathcal{F} & \cup & \{\alpha \subseteq subtypes(C)\} \cup \{\alpha \subseteq \varphi\} \\
 & & \\
\mathcal{F} & \cup & \{\{D\} \subseteq \alpha\} \cup \{\alpha \subseteq \varphi\} \equiv \\
\mathcal{F} & \cup & \{\{D\} \subseteq \alpha\} \cup \{\alpha \subseteq \varphi\} \cup \{\{D\} \subseteq \varphi\} \\
 & & \\
\mathcal{F} & \cup & \{\{D\} \subseteq \varphi\} \cup \{(subtypes(C) \cap \varphi) \subseteq \alpha\} \equiv \\
\mathcal{F} & \cup & \{\{D\} \subseteq \varphi\} \cup \{(subtypes(C) \cap \varphi) \subseteq \alpha\} \cup \{\{D\} \subseteq \alpha\} \\
 & & \quad \text{if } D \leq C
\end{array}
$$

Figure 6: Subtype Constraint Reduction

In the standard parlance, our inference system is monomorphic: it is field-insensitive and context-insensitive. Context- and field-sensitivity could be supported by adding class and method parameterization, as with Generic Java (GJ) [6].

3.4 Constraint Solving

Proxy inference generates constraints on the flow of proxies of the following forms (listed with the rules that generate them):

$T \leq U$	(I-If), (I-New)
$\alpha = subtypes(C) \cap \varphi$	(I-Cast)
$C_i \in \varphi \Rightarrow (\bar{S} \leq \bar{T}_i\ \&\ Q_i\ \varphi_i^C \leq \kappa\ \alpha^C)$	(I-Invoke)

Note that we represent the equality constraint from (I-Cast) as two subset constraints. Constraints on where coercions might be introduced have the form $\mathsf{proxy} \leq Q \Rightarrow l \in L$. Call the set of flow constraints $\mathcal{F}$, and the set of coercion constraints $\mathcal{C}$. We can solve these constraints as follows.

We can reduce $\mathcal{F}$ by continuously applying the rewriting rules shown in Figure 6. These reduce compound constraints into simpler ones following the subtyping rules, and iteratively discharge the implication constraints when the left-hand-side of the implication can be solved. When finished, all constraints will have the following forms: $C \leq D$, $\varphi \subseteq \varphi'$, and $Q \leq Q'$. The first form are subtyping requirements determined by the program; if they do not hold then the program would not be type-correct in FJ.

The remaining two forms can be solved by standard techniques. In particular, the qualifier constraints in $\mathcal{F}$ form an *atomic subtyping constraint system*. Given n such constraints, the fact that proxy and nonproxy form a finite lattice allows us to solve them in $O(n)$ time [36]. The set-type constraints in $\mathcal{F}$ are subset constraints, as occur in Andersen-style points-to analysis. Given n such constraints, these can be solved in at worst $O(n^3)$ time [2], though in practice it is often faster.

A solution σ to constraints in $\mathcal{F}$ is a mapping from qualifier variables κ to constants proxy and nonproxy, and set-type variables α to sets of class names $\{C_1, \ldots C_n\}$. The solution ensures that for each constraint $Q_1 \leq Q_2 \in \mathcal{F}$ we have $\sigma(Q_1) \leq \sigma(Q_2)$, and similarly for set-type constraints. We write $\sigma \models \mathcal{F}$ if σ is a solution of $\mathcal{F}$. We are interested in a *least* solution to α for set-types, to reduce spurious

$$
\begin{array}{lcl}
\mathcal{T}[\![\textbf{class } C \textbf{ extends } D \ \{ \ \bar{T} \ \bar{f}; K \ \bar{M} \ \}]\!] & \Rightarrow & \textbf{class } C \textbf{ extends } D \ \{ \ \sigma(\bar{T}) \ \bar{f}; \mathcal{T}[\![K]\!] \ \mathcal{T}[\![\bar{M}]\!] \ \} \\
\mathcal{T}[\![C(\bar{T} \ \bar{g}, \bar{S} \ \bar{f}) \ \{ \ \textbf{super}(\bar{g}); \ \textbf{this}.\bar{f} = \bar{f}; \ \}]\!] & \Rightarrow & C(\sigma(\bar{T}) \ \bar{g}, \sigma(\bar{S}) \ \bar{f}) \ \{ \ \textbf{super}(\bar{g}); \ \textbf{this}.\bar{f} = \bar{f}; \ \} \\
\mathcal{T}[\![S \ m(\bar{T} \ \bar{x}) \ \{ \ \textbf{return } e; \}]\!] & \Rightarrow & \sigma(S) \ m(\sigma(\bar{T}) \ \bar{x}) \ \{ \ \textbf{return } \mathcal{T}[\![e]\!]; \}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{T}[\![x]\!] & \Rightarrow & x \\
\mathcal{T}[\![\textbf{let } x = e_1 \textbf{ in } e_2]\!] & \Rightarrow & \textbf{let } x = \mathcal{T}[\![e_1]\!] \textbf{ in } \mathcal{T}[\![e_2]\!] \\
\mathcal{T}[\![e.f_i]\!] & \Rightarrow & \mathcal{T}[\![e]\!].f_i \\
\mathcal{T}[\![e.m(\bar{e})]\!] & \Rightarrow & \mathcal{T}[\![e]\!].m(\mathcal{T}[\![\bar{e}]\!]) \\
\mathcal{T}[\![\textbf{new } C(\bar{e})]\!] & \Rightarrow & \textbf{new } C(\mathcal{T}[\![\bar{e}]\!]) \\
\mathcal{T}[\![(N)e]\!] & \Rightarrow & (\sigma(N))\mathcal{T}[\![e]\!] \\
\mathcal{T}[\![\textbf{if } e_1 = e_2 \textbf{ then } e_3 \textbf{ else } e_4]\!] & \Rightarrow & \textbf{if } \mathcal{T}[\![e_1]\!] = \mathcal{T}[\![e_2]\!] \textbf{ then } \mathcal{T}[\![e_3]\!] \textbf{ else } \mathcal{T}[\![e_4]\!] \\
\mathcal{T}[\![\textbf{makeproxy } e]\!] & \Rightarrow & \textbf{makeproxy } \mathcal{T}[\![e]\!]
\end{array}
$$

$$
\mathcal{T}[\![\mathcal{E}^l]\!] \Rightarrow \begin{cases} \textbf{coerce } \mathcal{T}[\![\mathcal{E}]\!] & l \in L \\ \mathcal{T}[\![\mathcal{E}]\!] & \textit{otherwise} \end{cases}
$$

Figure 7: Transforming a FJ^i_Q expression to a FJ_Q expression following inference

constraints on qualifiers, and favor proxy over nonproxy for unconstrained qualifier variables so that we might delay inserting a coercion until absolutely necessary.

Given a solution σ to constraints $\mathcal{F}$, we can solve the coercion constraints $\mathcal{C}$. In particular, we apply σ to the left-hand-side of each implication in $\mathcal{C}$, and then solve. The result is a set L of all program labels that require a runtime coercion to properly typecheck. We write $\sigma, L \models \mathcal{C}$ for the set L and substitution σ that satisfies constraints $\mathcal{C}$.

3.5 Transformation

We can now transform a FJ^i_Q program to a FJ_Q program, using L and σ resulting from inference. FJ_Q differs from FJ^i_Q only in the addition of expressions of the form **coerce** e, and in the absence of all qualifier and set-type variables (these are substituted out by their solutions). The expression **coerce** e takes a possible proxy e, and coerces it to a non-proxy at runtime. Like **makeproxy** e, our semantics treats coercions generically, merely changing the tag on e to be nonproxy.

The transformation is shown as the function $\mathcal{T}[\![\cdot]\!]$ in Figure 7, where L and σ are "global" to avoid clutter. This function simply inserts coercions where directed by L, and rewrites the types on method declaration parameters and field declarations as directed by σ. To avoid clutter, it strips off all labels l.

In the case that we are doing a completely static analysis, e.g., to look for transparency violations, the fact that L is non-empty would denote a possible violation, so the transformation stage would signal an error, as directed by the user.

3.6 Properties

We wish to prove that FJ_Q is sound with respect to an operational semantics, and that a transformed FJ^i_Q program is sound with respect to the semantics of FJ_Q. For the first, the proof follows the standard syntactic approach of using *progress* and *preservation* lemmas. The second is done by proving well-typedness of the transformed program given the well-typedness of the source FJ^i_Q program.

Well-typedness of FJ_Q programs is expressed as the judgment $\vdash CL$ for class definitions, $\vdash M$ for method definitions and $\Gamma \vdash e : T; \Gamma$ for expressions. The typing rules are in Figure 9 in the Appendix. Typechecking FJ_Q is straightforward, and similar to inference on FJ^i_Q.

The operational semantics of FJ_Q is set up as an abstract machine. *Programs* consist of a store S and an expression to evaluate e, and the transition relation $\rightarrow$ maps programs (S, e) to programs (S', e'). The store maps variables x (either source program variables or fresh "addresses" allocated during evaluation) to values. The complete transition rules are presented in the Appendix (Figure 10). We also extend FJ_Q typing to programs, to support the proof of preservation.

The *progress* and *preservation* lemmas for FJ_Q are as follows:

LEMMA 3.1 (PROGRESS). *Given that* $\vdash (S, e) : T; \Gamma'$, *then either*

- e *is a variable* x.
- $(S, e) \rightarrow (S', e')$ *for some* S' *and* e'.
- (S, e) *is stuck due to a failed dynamic downcast.*

LEMMA 3.2 (PRESERVATION). *Given that* $\vdash (S, e) : T$, *and that* $(S, e) \rightarrow (S', e')$, *then* $\vdash (S', e') : U$ *such that* $U \leq T$.

Note that the type U of the program after it takes a step may be a subtype of its original type T due to both coercions (to downcast the proxy qualifier) and dynamic downcasts.

Using the above lemmas, the following theorem follows.

THEOREM 3.3 (TYPE SOUNDNESS). *Given* $\vdash e : T; \Gamma'$, *then either*

- $(\emptyset, e) \rightarrow^* (S, x)$ *for some* S *and* x.
- $(\emptyset, e) \rightarrow^* (S, e')$ *for some* S *and* e', *where* (S, e') *is stuck due to a failed dynamic downcast.*
- $(\emptyset, e)$ *executes forever.*

Here, we define $\rightarrow^*$ to mean the reflexive, transitive closure of the transition relation $\rightarrow$. Implicit in all of these statements is the presence of the well-formed class table CT. As is standard, the proofs of progress and preservation are by induction on the typing and evaluation derivations, respectively, and type soundness follows from them.

Finally, we can show that our proxy transformation from FJ^i_Q to FJ_Q is sound.

THEOREM 3.4 (INFERENCE SOUNDNESS). *Given a substitution* σ, *label set* L, *and an inference derivation* $\vdash_i CT$ *which generates constraints* $\mathcal{F}$ *and* $\mathcal{C}$, *if* $\sigma \models \mathcal{F}$ *and* $\sigma, L \models \mathcal{C}$, *then* $\vdash \mathcal{T}[\![CT]\!]$. *Moreover, for each subderivation of* $\vdash_i CT$ *which contains subderivations of the form*

1. $\vdash_i CL$

2. $\vdash_i M$

3. $\Gamma \vdash_i \mathcal{E}^l : T; \Gamma'$, *or*
   $\Gamma \vdash_c \mathcal{E}^l : T; \Gamma'$

there is a corresponding subderivation of $\vdash \mathcal{T}[\![CT]\!]$ *having the form:*

1. $\vdash \mathcal{T}[\![CL]\!]$

2. $\vdash \mathcal{T}[\![M]\!]$

3. $\sigma(\Gamma) \vdash \mathcal{T}[\![\mathcal{E}^l]\!] : \sigma(T); \sigma(\Gamma')$

The proof is by induction on the inference derivation. All proofs can be found in our companion technical report [34].

3.7 Discussion

Compared to past work in flow-sensitive type qualifiers, flow-sensitivity in our system is significantly simpler. The approach of Foster et al. [16] allows arbitrary memory locations to be treated flow-sensitively, which is complicated by the combination of aliasing and mutation. In particular, allowing the qualifier of a value to change flow-sensitively requires proving that the value is not aliased (is "linear"). In contrast, our approach only treats *local variables* flow-sensitively, and since Java has no "address-of" operator `&`, the contents of a local variable can only be accessed through that variable. Thus, we get linearity "for free," trading expressive power for simplicity. The caveat is that the implementation of **coerce** `x` provided by the user must only operate on the *variable* `x`, not on the *object* `x` refers to. For wrapper proxies, this is what happens: `x` is overwritten to point to the underlying object instead of the wrapper. If coercions do not meet this criteria, then they are not treated flow-sensitively.

It is because we are flow-sensitive only for local variables that we opted not to model field and variable updates in the FJ_Q. While adding updates would be straightforward (it is modeled in MJ [5] and existing qualifier systems [15, 16], for example), it would not change the character of our approach, adding only unnecessary complication.

In order to be able to support the full Java language, we had to address the use of the JNI and reflection mechanisms. We have assumed a conservative approach in this case, by demanding that no proxy object ever flows in the JNI API or in any reflection invocation. This approach inserts claims at all places where a native method is called, and on the arguments of `java.lang.reflect.Method.invoke(...)`, ensuring that no proxy Object will ever be passed to a reflection or JNI invocation. In addition, a proxy object might be accessed via reflection, e.g., by reading the fields of another object. Therefore, the analysis treats all objects obtained that way as possible proxies.

3.8 Other Applications

While the formal presentation of our analysis is specific to proxies, our added support for coercions can easily be folded into more general qualifier systems, admitting new or improved applications. Here we consider three possibilities.

Security-sensitive Data. Shankar et al. [37] describe an application of type qualifiers in which untrusted data, e.g., arriving from a user login prompt or a network connection, is given the qualifier tainted, while trusted data is given qualifier untainted. Qualifier inference is used to ensure that tainted data does not flow to functions requiring untainted data. A similar analysis is supported in Perl programs, except that checks for tainted data are performed dynamically. This has the drawback of the potentially-significant added runtime overhead of dynamic checks, but has the benefit that it is precise, and will thus avoid the false alarms generated by the purely static approach.

We can use our framework to implement a blending of these two approaches. In particular, the *pspec* would specify which routines returned tainted data, and which expected untainted data, while the *ispec* would implement coercions as a check to determine whether the data came from an untrusted source, e.g., by reading a required field from the object. This approach blends the two prior approaches by using static analysis to avoid many, but not all, runtime checks.

Stack allocation. Java objects have dynamic, unrestricted lifetimes, implemented using heap allocation and garbage collection. While stack-allocating objects could improve performance, avoiding dangling pointers would entail that no stack-allocated object escape its defining scope. This could happen if the object was assigned to a field or returned from its defining function. One solution would be to copy a stack-allocated object to the heap at the point it escapes its scope. However, doing so might violate transparency if that object's identity had already been revealed, e.g., by using the stack-allocated object as an argument to `==`.

Our analysis can support transparent stack allocation using two qualifiers, heap and stack, where the latter annotates an object that could be either heap- or stack-allocated, and the former indicates an object that must be heap-allocated; we thus have heap $\leq$ stack. Any operation that could reveal the identity of an object or cause it to escape (e.g., by assigning it to a field of a heap object) would require the object have qualifier heap. A coercion would check if an object was on the stack (perhaps using a bit mask), and copy it to the heap if necessary.

Not-null types. Another application is the use of null and nonnull qualifiers to characterize objects that are possibly null, or definitely not null, respectively [13]. This would provide a simple way of specifying the standard null-check elimination optimization as a qualifier system, and would allow users to manually annotate fields or method arguments as being nonnull, to avoid explicit null tests.

To implement this in our framework, the *pspec* would indicate that all occurrences of the constant `null` have qualifier null (including default initialization of fields), and that concrete object usages, e.g., to call a method, require that the object qualifier be nonnull. The *ispec* would implement coercions as null-checks (throwing an exception on failure), with flow-sensitivity naturally eliminating redundant checks. Of course, to be truly useful, we would require the cooperation of the JVM to avoid checks proven redundant by our framework.

4. ASYNCHRONOUS METHOD CALLS

Having described our proxy framework formally, we now describe our implementation of asynchronous method invocations in Java.

4.1 Framework Implementation

Our analysis is implemented as an extension to Soot [40] (version 2.1.0), a framework for analyzing and transforming Java classfiles. Soot provides a framework for implementing flow-insensitive points-to analyses called SPARK [27]. We extended SPARK to track proxies and generate set types based on points-to information. SPARK's constraint graph representation uses a node (corresponding variously to a qualifier variable κ or a set type variable α) for each local variable and method parameter. We extended this to be flow-sensitive by assigning multiple nodes to each variable or method parameter, one per use. As an optimization, we do so only for nodes that could possibly contain proxies, as determined by a flow-insensitive analysis. This reduces the total nodes to consider, since proxies are typically used sparingly in the program (relative to the total number of objects). For the applications presented in Section 5, this optimization yields a 6% to 45% improvement in the cost of the flow-sensitive analysis. Note that SPARK also supports context-sensitivity, but we have not taken advantage of this as of yet.

Programmers implement the *pspec* and *ispec* by providing three classes and linking them into the analysis:

1. The `AsyncGen` class in the *pspec* defines syntactic patterns that indicate where proxies are introduced. These patterns must, of course, be legal Java syntax that could have been compiled to bytecode.
2. The `Policy` class in the *pspec* defines coercions using a visitor over the Jimple syntax tree that specifies which expressions require non-proxies.
3. The `ClaimTransformer` class implements the *ispec*. It defines how call sites that create proxies are transformed, and how coercions are implemented. It may direct that supporting classes be linked into the transformed application.

Because Jimple represents typed bytecode, coercions that assign back to the original variable must be well-typed. Thus we give type `Object` to each Jimple variable x of type A that could contain a proxy. Whenever x is coerced, we assign the result to a newly-introduced variable `y` with type A, and replace with `y` subsequent occurrences of `x` in the continuation. This transformation is sound because proxies are treated transparently, and because there is no way to alias and mutate the storage of the original variable `x`.

4.2 Asynchronous Invocations

Programmers invoke methods asynchronously using the syntax

r `= Async.invoke(`t`,`$o.m(e_1,e_2,...)$`);`

According to the *pspec*, this syntax indicates that method m should be invoked asynchronously and the result (if any) returned to the caller will be a future. The method's arguments $e_1, e_2, ..., e_n$ are still evaluated in the current thread.

The *ispec* defines the steps needed to implement an asynchronous call. First, the program creates an anonymous subclass of `ProxyImpl` that encapsulates the invocation of method m. `ProxyImpl` has the following signature:

```
public class ProxyImpl implements Runnable, Wrapper {
  public void run();   // executes the invocation
  public Object get(); // acquires the result
}
```

The `Wrapper` interface simply defines a single `get` method, which extracts the underlying object for which the wrapper is a proxy.

```
public interface Wrapper { Object get(); }
```

Next, the `ProxyImpl` object is passed to a thread manager. Thread managers implement the Java 1.5 `Executor` interface:

```
public interface Executor {
  void execute(Runnable command);
}
```

The thread manager will call the `ProxyImpl`'s `run` method in a separate thread to achieve asynchrony. The `run` method will execute the method invocation $o.m(e_1,e_2,...)$ and store the result in a private field, to be extracted by a call to `get`. Finally, the `ProxyImpl` is returned to the caller of the original method m in place of the result r.

If the analysis determines that a program variable `x` with type `A` could contain a future, a *coercion* is required before `x` can be used concretely. The *ispec* implements coercions with the following code fragment:

```
(A)(x instanceof Wrapper ? ((Wrapper)o).get() : x)
```

That is, if `x` is a wrapper, then we must call `get` to extract the result. The `get` method in turn will `wait` if the result is not yet available.

Any implementation of `Executor` can be used as a thread manager. We have used the Java 1.5 `ThreadPoolExecutor`, which provides an extensible thread pool implementation, as well as our own `ThreadPerObjectExecutor`, which emulates *active objects* [26] by mapping each object receiving an asynchronous method call to an executor.

Note that programmers can influence where claims occur by performing "null" casts. That is, the expression $(C)e$ requires e's qualifier to be `nonproxy`, so casting it to its known type will have the effect of forcing a claim.

This design is both lightweight and flexible. Programmers can easily experiment with method asynchrony without rewriting substantial amounts of code. In addition, programmers can experiment with a variety of threading policies by choosing different thread managers.

A simple extension supports lazy evaluation. To invoke method m of object o lazily, the programmer uses the syntax:

r `= Lazy.invoke(`$o.m(e_1,e_2,...)$`);`

A `ProxyImpl` subclass is generated as above, but here the `run` method is called by `get` (called when the wrapper is claimed) if no final result yet exists.

4.3 Exceptions

If an asynchronous method call $o.m()$ throws an exception E, that exception is cached inside the future returned by m. When the future is claimed, the exception E is re-thrown.[2] This presents some challenges to the analysis.

[2]This differs from a `Future` in `util.concurrent`, whose `get` method declares it could throw an `ExecutionException`, encapsulating any exception thrown by the computation. As such, the programmer is required to handle `ExecutionException` each time that a future is claimed. Our implementation of claim essentially catches this exception, and then re-throws the exception it encapsulates.

The fact that claims could throw exceptions can be modeled as a simple extension to FJ_Q. We first must extend the language to model exceptions. We extend expressions e to include the form **try** e **catch** $E \Rightarrow e$, where E is the name of the exception being handled. Method declarations are extended to include **throws** clauses. We also add a form **throw** E for throwing an exception of type E (**throw** could take arbitrary expressions of exception type, but this simplifies the presentation). We extend the typing judgment from Figure 9 to include the *throw set* $\mathcal{T}$ of exceptions E that could be thrown by evaluating an expression.

The typing rule for try-blocks is:

$$\frac{\begin{array}{c}\Gamma, \vdash e_1 : T_1; \Gamma_1; \mathcal{T}_1 \qquad \Gamma \vdash e_2 : T_2; \Gamma_2; \mathcal{T}_2 \\ T_2 \leq T \qquad T_1 \leq T \qquad \Gamma' = \mathit{merge}(\Gamma_1, \Gamma_2) \\ \mathcal{T}' = \mathit{handles}(E, \mathcal{T}_1) \cup \mathcal{T}_2\end{array}}{\Gamma \vdash \textbf{try}\ e_1\ \textbf{catch}\ E \Rightarrow e_2 : T; \Gamma'; \mathcal{T}'}$$

The function $\mathit{handles}(E, \mathcal{T}_1)$ prunes those exceptions $E' \in \mathcal{T}_1$ which are subtypes of E. The resulting throw set is this pruned set and the set from the handler. This rule conservatively assumes any flow-sensitive effects of e_1 reflected in Γ_1 will not be seen in e_2. When checking a method consisting of expression e, we make sure that e's resulting throws set is covered by the **throws** clauses the method declares.

Now we must reflect into a proxy's type what exceptions it might throw. To do this we expand the proxy qualifier into a family of qualifiers, where each mentions an exception E that could be thrown if the qualified value is coerced. These form a lattice based on the subtyping relationship between exceptions E. For example, we have $\mathsf{proxy}E \leq \mathsf{proxy}E_2$ if $E \leq E_2$. For all E, we have $\mathsf{proxy} \leq \mathsf{proxy}E$.

The rule for **makeproxy** e becomes

$$\frac{\Gamma \vdash e : \mathsf{nonproxy}\ N; \Gamma'; \mathcal{T} \qquad E = \mathit{lub}(\mathcal{T})}{\Gamma \vdash \textbf{makeproxy}\ e : \mathsf{proxy}E\ N; \Gamma'; \emptyset}$$

That is, the exceptions that e could throw are reflected into its qualifier. For this rule to be sound, we must modify the operational semantics to capture any exception thrown when evaluating e in the proxy, and then re-throw the exception when doing the coercion. The typing rule for **coerce** reflects that an exception could be thrown:

$$\frac{\Gamma \vdash e : Q\ N; \Gamma'; \mathcal{T} \qquad Q \leq \mathsf{proxy}E}{\Gamma \vdash \textbf{coerce}\ e : \mathsf{nonproxy}\ N; \Gamma'; \mathcal{T} \cup \{E\}}$$

In our implementation, we must extend the definition of the `Wrapper` interface to define `get` methods that could throw the various expressions E determined by the analysis, and adjust `ProxyImpl` and claim code accordingly (which is easy to do automatically).

Given this formulation, we ensure that proxy inference deals with exceptions properly in a couple of ways. In the simplest case, we ensure that in expression **makeproxy** e, e never throws an exception. This is done by allowing the programmer to provide a handler for possible exceptions when creating the proxy. In particular, users can use an `Executor` that handles exceptions in a user-specified way inside spawned threads. This approach also requires that the user specify a "default" value for the object returned by a claim, since the swallowed exception will have prevented the method from returning a value. In our experience, this simple approach works fairly well in practice.

In the second case, we let inference determine where proxies could flow, signaling an error only if an inserted coercion could throw an exception not covered by the `throws` clause for the method in which it occurs. For many applications we have considered, unclaimed proxies do not flow outside the scope of a reasonable exception handler. This is frequently true for event-style server applications, which have an outermost exception handling block coupled with the event loop to catch exceptions raised by event handlers. In the case that a proxy does flow to an unexpected location, the user learns exactly where the offending claim was inserted and can manually alter the code to insert a handler. Alternatively, when the user specifies a method call should be asynchronous, she can provide a *handler object* whose `handle` method is called with argument E when a claim would cause E to be thrown. Any exceptions thrown by this handler (e.g., to delegate to an outer-scope handler) are reflected in the type of the proxy.

Even when the surrounding context can handle an exception E thrown due to a claim, it could be incorrect to do so. Some exceptions, like `IOException`, are thrown by many methods, and the exception generated by the claim may violate some invariant expected by the programmer. Though we have not yet done so, we should be able to ensure that a proxy can only throw to handlers that were present in its original context. To do this, rather than track the exceptions possibly thrown by an expression e, we could track all of the handlers that would catch exceptions thrown by e. These would create a similar partial order that would be folded into the proxy qualifier. At the same time, the typing judgment would keep track of the *handler context*, which is the set of all handlers that an exception could possibly throw to (including those in method callers). Typechecking a coercion would require that the handler context be a subset of the handlers mentioned in the proxy.

Note that all of this discussion need only apply to *checked* exceptions. As unchecked exceptions typically signal disastrous (unrecoverable) situations, we can choose to ignore them in the analysis.

4.4 Synchronization

Concurrent programs must balance safety and liveness, by guarding against invariant violations and preventing deadlock. Our approach no worse and no better than standard Java thread programming. When using asynchronous method calls, programmers must use ordering, synchronization, immutability, and other techniques to ensure safety and liveness—no automatic support is provided.

Ideally, ensuring a program is safe and live could be as lightweight as introducing an asynchronous invocation. In Lisp, this is trivial because programs are written in a mostly-functional (if not purely-functional) style, which means that added concurrency will not affect the program's safety. We contemplated approaches to inserting synchronization automatically [26, 7, 22, 17], but rejected this idea because of its lack of generality and potentially negative impact on performance. We discuss this issue more in Section 6.

Instead, we feel a more promising approach is to have programmers specify synchronization requirements declaratively. Declarative specifications should change infrequently, even as the programmer changes various method invocations to be or not be asynchronous. Therefore, the proper synchronization code could be generated from the specification as changes are made. Work in aspect-oriented programming [29, 25, 8] and language-level transactions [41, 19] aim to realize this goal. By not making any assumptions

test	tot (s)	per-check (ns)	% ovr
no claim	0.122	*n/a*	*n/a*
redundant claim	0.1637	16.37	34%
necessary claim	0.335	33.5	175%

Table 1: Overhead of inserted claims, $N = 10^7$

about synchronization, we can readily incorporate good results from these projects.

5. EVALUATION

We evaluate our framework in terms of (1) programming benefit (how does our framework simplify the programming task), (2) analysis effectiveness (how does it impact the runtime of the instrumented program), and (3) analysis performance (how fast is the analysis). We present a number of applications of both wrapper proxies and transparency checking to give a sense of the costs and benefits of our approach. As use of the Java 1.5 concurrency libraries becomes more widespread, we hope to adapt larger examples to use our framework.

We ran our experiments on a 2 GHz AMD Athlon 2600+ with 1 GB of RAM, running Mandrake Linux 9.1 (kernel version 2.4.21.)

5.1 Claim Overhead

Wrapper proxies can flow to potentially many parts of the program, and because our static analysis must be conservative, classes may be instrumented with redundant claims. To measure the performance overhead of necessary as well as redundant claims, we constructed a simple microbenchmark:

```
Object o,p = ...
for (int i = 0; i<N; i++) { p = o; p.m(); }
```

The method `m` simply increments a volatile counter. We varied `o` to be either a normal object, an already-claimed wrapper proxy, or an unclaimed wrapper proxy (in this last case, the copy from `o` to `p` ensures it will be claimed each time, since `o` never gets claimed and thus will never be rewritten to be the wrapped object). The results are shown in Table 1 for $N = 10^7$ (other values of N showed a similar relationship).

Redundant claims consist of essentially three instructions at runtime: an `instanceof` check, a cast, and an assignment. Our measurements show this adds 34% to the loop running time. Necessary claims require an additional synchronized method call and assignment, and cost more. However, these are unlikely to appear frequently because the future is overwritten after the underlying object is acquired, inducing only redundant claims from then on. In actual applications we expect the overhead of claims to be small because (1) not all method calls require claims, and (2) method calls perform real work, dwarfing the cost of claims relative to program running time.

5.2 Programming with Futures

A central benefit of our approach over a manual coding of proxies is that it simplifies the programming process. To illustrate this, we take an example from the `util.concurrent` API documentation [24, 12] that describes how to convert a "blocking service" into a non-blocking service using futures. The blocking service implements the following interface:

```
interface BlockingService {
  public Response serve (Request req)
    throws ServiceException;
}
```

We first present how we would convert `BlockingService` objects to be non-blocking using our approach, and then present the manual approach proposed in the documentation for `util.concurrent`.

Our Approach. Given `BlockingService` object `bs`, we make calls to its `serve` method asynchronous by simply changing existing method calls

```
bs.serve(request)
```

to be

```
Async.invoke(executor,bs.serve(request))
```

The analysis will infer where claims are required and insert them directly into the bytecode of both applications and library classes, based on user input. Assuming claims occur where `ServiceException`s can be caught, we are finished. Otherwise, we can modify invocations to include a wrapping exception handler, or add handlers to claim locations, as described in Section 4.3. We might also wish to insert "null casts" to force claims early, for performance reasons.

Manual Approach. To use Java 1.5 `Future`s instead, we would take the following steps [12]. First, we define a non-blocking variant of the `BlockingService` interface whose `serve` method returns a `Future`, and then build an adapter class to wrap a `BlockingService` object, as shown in Figure 8. The `serve` method of `NBSAdapter` creates a `task` to invoke the underlying `BlockingService` object's `serve` method, handling any exception locally. This task is executed by the adapter's `Executor` object after turning it into a `FutureTask`, which implements `Future`. The future is then returned to the caller.

Now we can make our original `bs` object non-blocking by creating `nbs = new NBSAdapter(bs)`. Existing calls

```
bs.serve(request)
```

are changed to be

```
nbs.serve(request)
```

At this point, we must adjust old client code to handle the fact that `nbs.serve` returns a `Future<Response>` rather than a `Response`. So that futures are claimed as late as possible, we must follow how `Response` objects would have flowed from calls to `serve` and sprinkle claims just before a futurized `Response` object is used. This can be tricky if `Response` objects were stored in containers that could be accessed by many methods or threads throughout the program. If a now-futurized `Response` object could flow into library routines or third-party components, the programmer may be forced to claim the future early, which could hurt performance.

Compared to one-line-per-invocation change imposed by our framework, this is a fair amount of programming overhead. Moreover, a similar overhead is required to undo the change.

5.3 Asynchronous RMI

For an asynchronous method call to be worthwhile, the added parallelism must overcome the added overheads, such as thread creation time and synchronization, to realize a

```
interface NonBlockingService {
  public Future<Response> serve (Request req);
}
class NBSAdapter implements NonBlockingService {
  public NBSAdapter (BlockingService svc) {
    this.blockingService = svc;
    this.executor = Executor.newFixedThreadPool(3);
  }
  public Future<Response> serve (final Request req) {
    Callable<Response> task = new Callable<Response>() {
      public Response call () {
        try {
          return blockingService.serve(req);
        }
        catch (ServiceException e) {
          e.printStackTrace();
          // more exception handling
        }
      }
    };
    FutureTask<Response> ftask =
      new FutureTask<Response>(task);
    executor.execute(ftask);
    return ftask;
  }
  private final BlockingService blockingService;
  private final Executor executor;
}
```

Figure 8: A `BlockingService` adapter class

performance gain. *Remote* method calls are a natural candidate, because they must pay the cost of a network round-trip time for each invocation. Indeed, asynchronous RPC was the initial motivation for Liskov and Shrira's promises [28], and recent work has considered the idea for Java [35, 39].

To illustrate this benefit, we have applied our framework to a RMI-based peer-to-peer service sharing application developed for a class at the University of Maryland[3]. Each peer can perform text processing using a number of composable *services*, which are simply references to objects implementing a `Service` interface. If the application does not have all of the services it wants, it can ask for them from the network, and will receive remote references for each in messages from peers. These are stored with the local services in a table.

The code to find a (potentially remote) service is roughly as follows:

```
Service findService(LocalPeer self, String sName) {
  Service s = self.getService(sName);
  if (s != null) return s;
  else {
    self.forward(new FindServiceMessage(sName));
    return getRemoteService(self, sName);
  }
}
```

If the service is present in the local table, the method immediately returns it. Otherwise, the `forward` method will use RMI to send messages to the node's peers, asking for the service. The first thing we did was make this method call asynchronous (though no future is returned)

The `getRemoteService` call will block (using `wait`) until it observes that the desired service has been installed in the table. This is problematic when the client application wishes to invoke `findService` n times to create a composed service as each call must wait until the prior service is found and the network will not be used to search for services in parallel. To address this issue, we made the call to `getRemoteService` *lazy*, changing it to be `Lazy.invoke(getRemoteService(self,sName))`. As this syntax introduces a wrapper proxy, the framework rewrites the caller's class to delay the invocation of the method until the proxy is unwrapped. Thus, all n calls to `getService` will proceed in parallel, and will only block when the service is used concretely.

[3] `http://www.cs.umd.edu/class/fall2003/cmsc433-0201/p5/p5.htm`

Analysis	Time	classes			
	(s)	analyzed	w/ fut.	transformed	claims
FI	73	1324	27	2	7
FS	92	1324	9	2	2
SPARK	66	1320	*n/a*	*n/a*	*n/a*

Table 2: Analysis Performance on Async RMI

Analysis Performance. The analysis times for this benchmark are shown in Table 2. Here we show the results of both our flow-sensitive analysis (FS), and a flow-insensitive variant of it (FI). Times are in seconds, and we show the total number of classes analyzed (which are largely library classes), those into which futures could flow, and finally those that were transformed. For those classes transformed, we indicate how many coercions (claims) were inserted. The results show the benefit of flow-sensitivity: fewer classes are polluted with futures and fewer claims are required.

The flow-sensitive analysis takes more time to run than the flow-insensitive version. Both process the same number of classes, but the flow-sensitive version generates more constraints. Indeed, the flow-insensitive analysis is virtually identical to the cost of just running the SPARK without modification. The flow sensitive analysis uses the result of a flow-insensitive analysis to limit the number of variables that are analyzed flow-sensitively. In particular, only the variables that have qualifier `proxy` are re-analyzed flow sensitively. However, the time saved compared to running the flow-sensitive analysis for the whole program is still significant, even though we are running the analysis twice.

Runtime Performance. To assess the runtime benefit to asynchronous remote invocations, we ran some simple experiments on a two-node network connected by 100 Mbps Ethernet. The application attempts to acquire n services, for $1 \leq n \leq 10$, all of which are non-local. We compare the original application (Orig) to our changed version (Async). In addition to normal RMI messaging, we ran a version that inserts an 80 ms delay for each message send, to simulate a wide area message. The results shown in Table 3 represent the median of 11 runs, with all times in milliseconds (the mean and median values were similar).

For the local area traffic, the added parallelism of asynchronous RMI nets performance gains of up to 40%. For the delayed case, the running time of the original application tracks the number of services times the round trip delay, while the Async version significantly amortizes this cost.

Of course, these results could have been achieved by rewriting the application by hand to capture the invocation, and

Version	Services requested and used									
	1	2	3	4	5	6	7	8	9	10
Orig.	18	32	48	65	90	100	113	124	143	153
Async	17	26	34	42	50	57	62	78	83	84
Orig. + delay	101	203	304	406	507	607	707	811	916	1010
Async + delay	106	117	122	133	145	153	157	160	171	178

Table 3: Elapsed time (s) of Peer-to-Peer RMI application with varying workload

acquire it before applying the result. Our framework made it significantly easier to do this: we only had to annotate two method calls, and the framework did the rest automatically.

5.4 Transparency Checking

We have also used our framework to search for possible transparency violations through the use of interface proxies. Here, we consider a programmer that might like to specialize an object implementing interface I, e.g., to count how often a particular method is called. Following the proxy design pattern, the programmer could use a *dynamic proxy class* [9] to create a method-counting object that also implements I, which forwards calls to the original object. Our framework can ensure that the program will never distinguish between the proxy and the underlying object by using an identity-related operation, like `==`, `instanceof`, etc. This is done with the following policy and implementation specification:

Policy Calls to `Proxy.newProxyInstance(...)` introduce proxies. All expressions that are identity-revealing must operate on non-proxies, including `synchronized`, `==`, and `instanceof`. Note that unlike futures and other wrapper proxies, method calls do not require the object be a non-proxy.

Implementation No code is needed to generate proxies (that is already being done by the program) or to coerce them. Any requirement of a coercion implies a possible transparency violation, which is signaled by the analysis.

We ran our checker on two examples: an XML-based implementation of SOAP over RMI that uses dynamic proxy classes [38], and the Soot bytecode analysis framework [40] (version 2.0.1).[4] In the former case, the analysis tracks all proxies created with `Proxy.newProxyInstance`. In the latter, we selected three different methods that return interfaces, and told the checker that calling these methods might return proxies. This simulates a user wishing to proxy an object returned by one of these methods, e.g., to perform profiling, but ensuring that transparency will not be violated.

We ran the flow sensitive and a flow insensitive analysis to detect possible errors. For the SOAP/RMI example, we ran the checker over the code as is, and found no transparency violations. Table 4 summarizes the results. Once again, the flow-insensitive analysis had essentially the same running time as SPARK points-to analysis (not shown), and the flow-sensitive version added some overhead. Interestingly, the flow-sensitive analysis adds no value in this case. It could potentially reduce false positives due to spurious flows, but does not do so.

[4]We analyze Soot 2.0.1 because analyzing 2.1.0 causes our benchmark machine to swap.

Version	Time (s)	# of classes		Errors
		analyzed	with proxies	
FI	48	2189	3	0
FS	58	2189	3	0
SPARK	45	2189	*n/a*	*n/a*

Table 4: Analysis Performance for SOAP/RMI

Example	Time (s)		# of classes		Errors
	FI	FS	analyzed	with proxies	
1	181	210	2092	16	0
2	174	209	2092	3	1
3	183	214	2092	12	9
SPARK	151	*n/a*	2092	*n/a*	*n/a*

Table 5: Analysis Performance for 3 Soot Examples

The Soot examples for the three different methods are shown in Table 5. The SPARK number is the average time for all three examples, which had similar running times. We looked at the reported violations, and verified that they were genuine transparency violations that could lead to bugs. Once again, it was interesting to see that flow-sensitivity added no precision (only overhead!), and that the original SPARK analysis times are relatively close to our flow-insensitive analysis times.

6. RELATED WORK

Proxies. Gamma et al. [18] present many uses of the proxy design pattern, including remote references, lazy evaluation, and access control. Other uses[5] include memoization, delegation, synchronization addition, generic event listeners, and views for abstract data types. Java's dynamic proxy classes [9] permit the simple construction of interface proxies, and have been used in a variety of applications [38, 4].

Static Analysis. Our analysis is a variant of qualifier inference, which draws upon techniques developed in other static analyses, including constraint-based analysis [1] and points-to analysis [10]. Our approach extends Foster et al.'s qualifier inference [15] with support for coercions that implement checks at runtime, e.g., to claim a future. These coercions are treated flow-sensitively. Foster et al. also define a flow-sensitive variant of their analysis [16], but their approach allows heap locations, and not just variables, to be treated flow-sensitively. This adds expressive power but significant complication.

[5]See, for example `http://blog.monstuff.com/archives/000098.html`.

Asynchronous Method Calls and Futures. The notion of a future was popularized by Halstead in MultiLisp [23]. In a dynamically-typed language like Lisp or Scheme, potentially any value could be a future, necessitating a runtime check. Flanagan and Felleisen [14] define a whole-program static analysis for reducing eliminating some unnecessary checks; our analysis conversely adds needed checks based on the possible flow of futures.

Liskov and Shrira proposed *promises* [28], which are futures for statically-typed languages. A promise is a type parameterized by the type of object it will ultimately compute, like a Java 1.5 `Future` [24]. We found a number of applications of futures to statically-typed, object-oriented languages [31, 24, 20, 35, 11].

Mandala [30] is another framework that provides asynchronous method invocation and futures for Java. Asynchronous calls are implemented with reflection, using dynamic proxy classes, which is more modular than our approach. However, Mandala is less efficient and less transparent. Identity-revealing operations like `==` could distinguish the proxied object. Dynamic proxies use reflection for each method call, which is notoriously slow (more than an order of magnitude slower than a normal method call on our benchmark machine). To work around this overhead, a Mandala programmer can treat the returned value of an asynchronous call as an explicit `FutureClient` object (much like a Java 1.5 `Future`) which must be manually claimed, thus sacrificing the programming benefit of transparency.

A number of languages support *active objects* [26], such as the SCOOP extension to Eiffel [7] and Io [22], which return futures.Method calls are handled by a per-object thread, and automatically synchronized based on programmer-supplied method preconditions. While simple to use, programmers are forced to use concurrency only on a per-object basis, as opposed to per activity, which could severely limit performance without potentially unnatural program restructurings.In our approach, concurrency is handled per-method by arbitrary `Executor` objects, but synchronization must be handled by the programmer.

Polyphonic C# [3] adds concurrency abstractions to C# based on the join calculus. Method declarations annotated as `async` are always invoked asynchronously. These methods must not return results, so there is no need for futures.

Asynchronous *remote* method invocation can be used to batch remote calls and thus amortize the delay of round-trip times. Promises were developed in this context. Raje et al. [35] propose an approach in which the returned future is made manifest to the programmer, adding to the programming burden. Sysala and Janecek [39] require that remote calls be provided a *callback*, to be invoked the result is available. This simplifies exception handling but obscures the control flow of the program, making debugging more difficult. It also forces programmers to distinguish between remote and local references, eliminating the transparency afforded by RMI.

7. CONCLUSIONS

We have presented a simple and flexible framework for transparent programming with proxies in Java. The framework uses a sound static analysis to track the flow of proxies throughout the program. The analysis is based on qualifier inference [15], with two extensions: we permit the use of dynamic coercions to allow proxies to have runtime effect, and use flow-sensitivity to avoid redundant coercions. We have used our framework to implement a natural form of asynchronous and lazy method invocation for Java, and to check for possible transparency violations when using the proxy design pattern. The framework is general enough to apply to other interesting applications, including the tracking of security-sensitive data, and supporting not-null types and stack-allocated objects.

We are currently pursing two avenues of future work. First, we are generalizing our framework to support arbitrary qualifiers, to support the other applications mentioned above. In doing so, we plan to support more sophisticated context-sensitive analysis. Second, we are exploring how to make our analysis incremental, to avoid reanalyzing the whole program each time a source file is changed. Rather, we are developing a dependency-tracking system that would allow for selective reanalysis of unchanged classes, possibly in the background for better performance. We would hope to generalize our approach to other static analyses.

Acknowledgments. We thank Jeff Foster, Nikhil Swamy, James Rose, and the anonymous referees for helpful comments on drafts of this paper.

8. REFERENCES

[1] Alexander Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming (SCP)*, 35(2):79–111, 1999.

[2] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[3] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C#. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2003. Special Issue of papers from FOOL 9.

[4] Gregory Biegel, Vinny Cahill, and Mads Haahr. A dynamic proxy based architecture to support distributed Java objects in a mobile environment. In *Proceedings of the CoopIS/DOA/ODBASE*, pages 809–826, 2002.

[5] Gavin M. Bierman, Matthew J. Parkinson, and Andrew M. Pitts. An imperative core calculus for Java and Java with effects. Technical Report 563, University of Cambridge Computer Laboratory, April 2003.

[6] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: adding genericity to the Java programming language. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, British Columbia, Canada, 1998.

[7] Michael James Compton. SCOOP: An investigation of concurrency in Eiffel. Master's thesis, Department of Computer Science, The Australian National University, December 2000.

[8] Xianghua Deng, Matthew B. Dwyer, John Hatcliff, and Masaaki Mizuno. Invariant-based specification, synthesis, and verification of synchronization in concurrent programs. In *Proceedings of the IEEE International Conference on Software Engineering (ICSE)*, pages 442–452, Orlando, Florida, USA, 2002.

[9] Dynamic proxy classes. `http://java.sun.com/j2se/1.5.0/docs/guide/reflection/proxy.html`. JDK 1.5 documentation.

[10] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 242–256, Orlando, Florida, USA, 1994.

[11] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: spatial and temporal selection of reification. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 27–46, Anaheim, California, USA, 2003.

[12] Executor examples. `http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/etc/notes/tim-execu%tor-examples.html?rev=1.5`.

[13] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 302–312, Anaheim, California, USA, 2003.

[14] Cormac Flanagan and Matthias Felleisen. The semantics of Future and its use in program optimizations. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 209–220, San Francisco, CA, January 1995.

[15] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 192–203, May 1999.

[16] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, Berlin, Germany, June 2002.

[17] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 212–223, Montreal, Canada, 1998.

[18] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[19] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 388–402, Anaheim, California, USA, 2003.

[20] Claude Hussenet. Personal Communication. Describes the use of JSR 166 futures for an enterprise application.

[21] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001.

[22] Io: A small programming language. `http://www.iolanguage.com/`.

[23] Robert H. Halstead Jr. Multilisp - a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):501–538, 1985.

[24] JSR 166: Concurrency utilities. `http://www.jcp.org/en/jsr/detail?id=166`.

[25] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersen, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, Budapest, Hungary, 2001. Springer-Verlag.

[26] R. Greg Lavender and Douglas C. Schmidt. Active object: an object behavioral pattern for concurrent programming. In *Proceedings of the Pattern Languages of Programs*, September 1995.

[27] Ondrej Lhoták and Laurie Hendren. Scaling Java points-to analysis using SPARK. In *Proceedings of the International Conference on Compiler Construction (CC)*, volume 2622 of *Lecture Notes in Computer Science*, pages 153–169, Warsaw, Poland, 2003.

[28] Barbara Liskov and Liuba Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 260–267, Atlanta, Georgia, USA, July 1988.

[29] Cristina Videira Lopes and Karl J. Lieberherr. Abstracting process-to-function relations in concurrency object-oriented applications. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 821 of *Lecture Notes in Computer Science*, pages 81–99, Bologna, Italy, July 1994. Springer.

[30] Mandala. `http://sourceforge.net/projects/mandala/`.

[31] Dragos A. Manolescu. Workflow enactment with continuation and future objects. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 40–51, Seattle, Washington, USA, 2002.

[32] Eric Mohr, David A. Kranz, and Robert H. Halstead Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, 1991.

[33] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *Proceedings of the International Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 66–77, La Jolla, California, USA, 1995.

[34] Polyvios Pratikakis, Jaime Spacco, and Michael Hicks. Transparent proxies for Java futures. Technical Report CS-TR-4574, Department of Computer Science, University of Maryland, College Park, March 2004.

[35] Rajeev R. Raje, Joseph I. William, and Michael Boyles. An asynchronous remote method invocation (ARMI) mechanism for Java. In *Proceedings of the ACM Workshop on Java for Science and Engineering Computation*, Las Vegas, Nevada, 1997.

[36] Jakob Rehof and Torben Mogensen. Tractable constraints in finite semilattices. *Science of Computer Programming*, 35(2–3):191–221, 1999.

[37] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th Usenix Security Symposium*, Washington, D.C., August 2001.

[38] Aleksander Slominski, Madhusudhan Govindaraju, Dennis Gannon, and Randall Bramley. Design of an XML based interoperable RMI system : SoapRMI C++/Java 1.1. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1661–1667, Las Vegas, Nevada, June 2001.

[39] Tomás Sysala and Jan Janecek. Optimizing remote method invocation in Java. In *Proceedings of the International Workshop on Database and Expert Systems Applications (DEXA)*, pages 29–35, Aix-en-Provence, France, September 2002.

[40] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot–a Java optimization framework. In *Proceedings of the IBM Centers for Advanced Studies Conference (CASCON)*, pages 125–135, 1999.

[41] Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Transactional monitors for concurrent objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Oslo, Norway, 2004.

APPENDIX

A. PROXY CALCULUS FJ_Q

Here we include more details on the explicitly-typed calculus FJ_Q, introduced in Section 3, including its typing rules and operational semantics.

A.1 Typing

The syntax of FJ_Q is the same as FJ^i_Q (Figure 1), minus qualifier and set type variables κ and α, plus expressions **coerce** e. The typing rules are shown in Figure 9. We have stripped labels from expressions for clarity, since they are not used. The subtyping rules and auxiliary definitions are the same as those in Figures 2 and 3.

The rules are basically straightforward analogues of the inference rules. Note that there are two rules for typing casts. The (Cast) rule types an upcast or a downcast, and the (SCast) rule types a "stupid" cast. The last is a technical device borrowed from FJ to allow all possible casts to be considered well-typed, which is necessary to prove type soundness via the property of type preservation (theorems are stated in Section 3.6). The Java compiler would reject programs containing stupid casts.

A.2 Operational Semantics

The operational semantics of FJ_Q are set up as an abstract machine. *Programs* consist of a store S and an expression to evaluate e, and the transition relation $\rightarrow$ maps programs (S, e) to programs (S', e'). We use a call-by-value *allocation-style* semantics [33], in which all objects are allocated and looked up in the store, rather than being substituted into the term. This allows us to model the flow-sensitivity of coercions on variables. The store essentially represents a hybrid of the stack and the heap. The complete transition rules are presented in Figure 10.

Since this is a qualified system, the store maps variables to *qualified store values*, which are store values h paired with a qualifier Q. A store value is simply an object of the form **new** $C(\bar{y})$, where the variables $\bar{y}$ index other qualified store values in S. Qualified store values are allocated by the following (TransAnnot) rule, which replaces a store value h with a fresh variable x, and then maps that variable to h in the store S:

$$(S, \textbf{new } C(\bar{y})) \rightarrow (S \uplus \{x \mapsto (\mathsf{nonproxy}, \textbf{new } C(\bar{y}))\}, x)$$

The other computation rules always operate on variables indexing the store, and so must "look up" the corresponding value for evaluation. For example, the (TransInvoke) rule is between two variables x and y; it looks up x in the store to discover a function, and then continues by evaluating the function's body e, having updated the store to map the function's parameter z to the actual argument pointed at by y.

$$\frac{S(x) \;=\; (\mathsf{nonproxy}, \textbf{new } C(\bar{y})) \qquad \mathit{mbody}(m, C) = (\bar{z}, e)}{(S, x.m(\bar{y})) \rightarrow (S \uplus \{\bar{z} \mapsto S(\bar{y})\}, e[\textbf{this} \mapsto x])}$$

Note that we encode freshness by not adding variables to the domain of the store if they are already present; this is illustrated by the use of $\uplus$. We can always enforce this condition using alpha conversion.

All qualified store values that are used concretely must have qualifier nonproxy, indicating that the actual value is available. These conditions match those in the type rules. Relaxing a requirement in the type rules (e.g., as would happen for interface proxies) would require relaxing it here.

The (TransCoerce) rule handles flow-sensitive coercions:

$$(S \uplus \{x \mapsto (Q, h)\}, \textbf{coerce } x) \rightarrow (S \uplus \{x \mapsto (\mathsf{nonproxy}, h)\}, x)$$

Here, when a variable x is coerced, we remap x in the output store so that its qualifier is nonproxy. Therefore, subsequent uses of x will not require coercions. This will have little effect unless x was a variable in the original program. Otherwise it was a constant expression, which will never again be reused. Note that the (TransCoerce) rule is well-defined for *all* qualified store values, not just those with qualifier proxy; this is critical because the subtyping rule nonproxy $\leq$ proxy employed by the type system allows non-proxies to be used wherever proxies are expected.

We extend the typing judgment to programs (S, e) as shown in Figure 9. Here, the (CheckState) rule requires that the store S can be characterized by a Γ sufficient to typecheck e. Notice that the (CheckStore) rule only checks values mapped to by variables in the domain of Γ, rather than the domain of S. This allows Γ to refer only to variables in the transitive closure of the variables appearing in e; any other indexes in the store are essentially garbage, and could be removed. Also note that (CheckNewQ) returns the exact (dynamic) type of objects that it finds. Because these objects could be given "higher" type in the program e, we allow $T \leq \Gamma(x)$ in the (CheckStore) rule.

$\boxed{\Gamma \vdash e : T; \Gamma'}$

$$\text{Var}\ \frac{}{\Gamma[x \mapsto T] \vdash x : T; \Gamma[x \mapsto T]} \qquad \text{Let}\ \frac{\Gamma \vdash e_1 : T; \Gamma_1 \quad \Gamma_1[x \mapsto T] \vdash e_2 : T'; \Gamma'[x \mapsto T'']}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : T'; \Gamma'}$$

$$\text{If}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : \mathsf{nonproxy}\ N_1; \Gamma_1 \quad \Gamma_1 \vdash e_2 : \mathsf{nonproxy}\ N_2; \Gamma_2 \\ \Gamma_2 \vdash e_3 : T_3; \Gamma_3 \qquad \Gamma_3 \vdash e_4 : T_4; \Gamma_4 \\ T_3 \le T \quad T_4 \le T \quad \Gamma' = \mathit{merge}(\Gamma_3, \Gamma_4)\end{array}}{\Gamma \vdash \textbf{if}\ e_1 = e_2\ \textbf{then}\ e_3\ \textbf{else}\ e_4 : T; \Gamma'} \qquad \text{Invoke}\ \frac{\begin{array}{c}\Gamma \vdash e_0 : \mathsf{nonproxy}\ N; \Gamma' \quad \Gamma' \vdash \bar{e} : \bar{S}; \Gamma'' \\ \mathit{mtype}(m, N) = \bar{T}_1 \to U_1, \ldots \bar{T}_n \to U_n \\ \bar{S} \le \bar{T}_i \qquad U_i \le T \qquad \text{for all } i\end{array}}{\Gamma \vdash e_0.m(\bar{e}) : T; \Gamma''}$$

$$\text{Field}\ \frac{\Gamma \vdash e : \mathsf{nonproxy}\ N; \Gamma' \quad \mathit{fields}(N) = \bar{T}\ \bar{f}}{\Gamma \vdash e.f_i : T_i; \Gamma'} \qquad \text{New}\ \frac{\mathit{fields}(\{C\}^C) = \bar{T}\ \bar{f} \quad \Gamma \vdash \bar{e} : \bar{S}; \Gamma' \quad \bar{S} \le \bar{T}}{\Gamma \vdash \textbf{new}\ C(\bar{e}) : \mathsf{nonproxy}\ \{C\}^C; \Gamma'}$$

$$\text{Cast}\ \frac{\begin{array}{c}\Gamma \vdash e : \mathsf{nonproxy}\ \varphi^D; \Gamma' \\ \varphi_1 = \mathit{subtypes}(C) \cap \varphi \quad \varphi_1 \neq \emptyset\end{array}}{\Gamma \vdash (C)e : \mathsf{nonproxy}\ \varphi_1^C; \Gamma'} \qquad \text{SCast}\ \frac{\begin{array}{c}\Gamma \vdash e : \mathsf{nonproxy}\ \varphi^D; \Gamma' \\ \varphi_1 = \mathit{subtypes}(C) \cap \varphi \quad \varphi_1 = \emptyset \\ \mathit{stupid\ warning}\end{array}}{\Gamma \vdash (C)e : \mathsf{nonproxy}\ \mathit{subtypes}(C)^C; \Gamma'}$$

$$\text{MakeProxy}\ \frac{\Gamma \vdash e : \mathsf{nonproxy}\ N; \Gamma'}{\Gamma \vdash \textbf{makeproxy}\ e : \mathsf{proxy}\ N; \Gamma'} \qquad \text{CoerceExp}\ \frac{\Gamma \vdash e : Q\ N; \Gamma' \quad e \neq x}{\Gamma \vdash \textbf{coerce}\ e : \mathsf{nonproxy}\ N; \Gamma'}$$

$$\text{CoerceVar}\ \frac{\Gamma \vdash x : Q\ N; \Gamma \quad \Gamma = \Gamma'[x \mapsto Q\ N]}{\Gamma \vdash \textbf{coerce}\ x : \mathsf{nonproxy}\ N; \Gamma'[x \mapsto \mathsf{nonproxy}\ N]}$$

$\boxed{\vdash M}$

$$\text{Method}\ \frac{\begin{array}{c}\bar{x} : \bar{T}, \textbf{this} : \mathsf{nonproxy}\ \{C\}^C \vdash e : U \quad U \le S \\ CT(C) = \textbf{class}\ C\ \textbf{extends}\ D\ \{\ \ldots; \ldots\ \} \\ \mathit{override}(m, D, \bar{T} \to S)\end{array}}{\vdash S\ m(\bar{T}\ \bar{x})\ \{\ \textbf{return}\ e;\}}$$

$\boxed{\vdash CL}$

$$\text{Class}\ \frac{\begin{array}{c}K = C(\bar{T}\ \bar{g}, \bar{S}\ \bar{f})\ \{\ \textbf{super}(\bar{g});\ \textbf{this}.\bar{f} = \bar{f};\ \} \\ \mathit{fields}(D) = \bar{T}\ \bar{g} \qquad \vdash \bar{M}\end{array}}{\vdash \textbf{class}\ C\ \textbf{extends}\ D\ \{\ \bar{T}\ \bar{f}; K\ \bar{M}\ \}}$$

$\boxed{\Gamma \vdash (Q, h) : T}$

$$\text{CheckNewQ}\ \frac{\mathit{fields}(\{C\}^C) = \bar{T}\ \bar{f} \quad \Gamma(\bar{x}) = \bar{U} \quad \bar{U} \le \bar{T}}{\Gamma \vdash (Q, \textbf{new}\ C(\bar{x})) : Q\ \{C\}^C}$$

$\boxed{\vdash S : \Gamma}$

$$\text{CheckStore}\ \frac{\Gamma \vdash S(x) : T \quad T \le \Gamma(x) \quad \text{all x} \in \text{dom}(\Gamma)}{\vdash S : \Gamma}$$

$\boxed{\vdash (S, e) : T}$

$$\text{CheckState}\ \frac{\vdash S : \Gamma \quad \Gamma \vdash e : T; \Gamma'}{\vdash (S, e) : T}$$

Figure 9: FJ_Q: Typing

$\boxed{(S, e) \to (S', e')}$

Transitions:

$$\text{TransAnnot}\ \frac{}{(S, \mathbf{new}\ C(\bar{x})) \to (S \uplus \{x \mapsto (\mathsf{nonproxy}, \mathbf{new}\ C(\bar{x}))\}, x)}$$

$$\text{TransInvoke}\ \frac{S(x) \ = \ (\mathsf{nonproxy}, \mathbf{new}\ C(\bar{y})) \qquad \mathit{mbody}(m, C) = (\bar{z}, e)}{(S, x.m(\bar{y})) \to (S \uplus \{\bar{z} \mapsto S(\bar{y})\}, e[\mathbf{this} \mapsto x])}$$

$$\text{TransField}\ \frac{S(x) \ = \ (\mathsf{nonproxy}, \mathbf{new}\ C(\bar{x})) \qquad \mathit{fields}(\{C\}^C) = \bar{T}\ \bar{f}}{(S, x.f_i) \to (S, x_i)}$$

$$\text{TransCast}\ \frac{S(x) \ = \ (\mathsf{nonproxy}, \mathbf{new}\ D(\bar{y})) \qquad D \leq C}{(S, (C)x) \to (S, x)}$$

$$\text{TransLet}\ \frac{}{(S, \mathbf{let}\ x = y\ \mathbf{in}\ e) \to (S \uplus \{x \mapsto S(y)\}, e)}$$

$$\text{TransIfTrue}\ \frac{}{(S, \mathbf{if}\ x = x\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2) \to (S, e_1)}$$

$$\text{TransIfFalse}\ \frac{x \neq y}{(S, \mathbf{if}\ x = y\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2) \to (S, e_2)}$$

$$\text{TransProxy}\ \frac{S(x) \ = \ (\mathsf{nonproxy}, h)}{(S, \mathbf{makeproxy}\ x) \to (S \uplus \{y \mapsto (\mathsf{proxy}, h)\}, y)}$$

$$\text{TransCoerce}\ \frac{}{(S \uplus \{x \mapsto (Q, h)\}, \mathbf{coerce}\ x) \to (S \uplus \{x \mapsto (\mathsf{nonproxy}, h)\}, x)}$$

Congruence rules:

$$\text{C-CongruenceE}\ \frac{(S, e) \to (S', e')}{\begin{array}{c}(S, e.m(\bar{y})) \to (S', e'.m(\bar{y})) \\ (S, e.f_i) \to (S', e'.f_i) \\ (S, (N)e) \to (S', (N)e') \\ (S, \mathbf{let}\ x = e\ \mathbf{in}\ e_2) \to (S', \mathbf{let}\ x = e'\ \mathbf{in}\ e_2) \\ (S, \mathbf{if}\ e = e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3) \to (S', \mathbf{if}\ e' = e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3) \\ (S, \mathbf{if}\ x = e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2) \to (S', \mathbf{if}\ x = e'\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2) \\ (S, \mathbf{makeproxy}\ e) \to (S', \mathbf{makeproxy}\ e') \\ (S, \mathbf{coerce}\ e) \to (S', \mathbf{coerce}\ e')\end{array}}$$

$$\text{C-CongruenceBarE}\ \frac{(S, \bar{e}) \to (S', \bar{e}')}{\begin{array}{c}(S, \mathbf{new}\ C(\bar{e})) \to (S', \mathbf{new}\ C(\bar{e}')) \\ (S, x.m(\bar{e})) \to (S', x.m(\bar{e}'))\end{array}}$$

Figure 10: FJ_Q: Operational Semantics