Transparent Proxies for Java Futures

Polyvios Pratikakis
polyvios@cs.umd.edu
University of Maryland
at College Park
College Park, MD 20740

Jaime Spacco
jspacco@cs.umd.edu
University of Maryland
at College Park
College Park, MD 20740

Michael Hicks
mwh@cs.umd.edu
University of Maryland
at College Park
College Park, MD 20740

CS-TR-4574, UMIACS-TR-2004-19

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 [19]. 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, which employs a static analysis based on qualifier inference [16], 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, and 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.

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 [27]. In MultiLisp, the syntax `(future` *e*`)` designates that expression *e* should be evaluated concurrently. Since the result is not immediately available, the current thread is given a placeholder for it, called a future. Because MultiLisp is dynamically typed, the runtime system can manipulate futures *transparently*; the programmer is not required to write code to wait for and extract the actual result from the future.[1] This makes the use of futures simple and lightweight.

In contrast, proposals to add futures to statically-typed languages [32, 28, 37] make futures manifest to the programmer. For example, JSR 166 [28] defines a future with the following Java interface:

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

The programmer must *claim* the underlying *V* manually with `get`, which can require considerable programming effort when adapting a program to use futures (discussed in detail in Section 5.2).

Moreover, futures are not transparent, since they are represented as an object separate from their contents. Thus a programmer could distinguish a future from its underlying object unintentionally. For example, a Java programmer could mistakenly store both in the same `Set` because the default `Object.equals` method uses `==` for comparisons, and typical `equals` methods use `instanceof`, both of which depend on an object's identity. Avoiding these situations requires a careful understanding of how futures and their underlying objects could interact in a program.

Motivated to support futures in Java with the same ease-of-use and transparency as dynamically-typed MultiLisp, we have developed a framework to simplify proxy programming in general. At the core of the framework is a static analysis that tracks how a proxy might flow through the program, coupled with a transformation that implements how proxies are manipulated at runtime. We have used our framework for a variety of applications:

- We have implemented support for asynchronous method calls which could return futures. Adding or removing asynchrony is extremely lightweight, as the framework generates the code for spawning or delegating to threads as well as the code for managing futures returned as results. Programmers can influence performance by specifying a *thread manager* to handle an invocation, including global thread pools, per-object thread pools, and others. Programmers can also influence where futures are claimed. In essence, the framework drastically simplifies programming

[1] There is the possible need for synchronization due to side effects in *e*.

with JSR 166 futures, which is timely given JSR 166's impending inclusion in the Java standard.

- A future is an example of a *wrapper* proxy, in which a container wraps an underlying object. We have also implemented support for lazy evaluation in Java using *suspensions*, which are wrapper proxies that encapsulate lazy computations. 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.

- *Interface proxies* share an interface with their target object, as specified by the proxy design pattern [19]. As with wrapper proxies, incorrect usage of these proxies could violate transparency. Section 5.4 describes how we used our framework to discover possible transparency violations arising from the introduction of interface proxies in large programs.

Our static analysis is based on *qualifier inference* [16], 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 coercing 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 3.7, they enable a number of new or improved applications, including tracking security-sensitive data in a program [41], and supporting 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) [25], 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 3.7).

- We present a design and implementation of asynchronous and lazy method invocations for Java. Because handling of futures is automatic, our approach is substantially easier to use than approaches that require manipulating futures manually [32, 28, 37]. At the same time, performance can be easily tuned by the programmer. Our approach combines nicely with other Java features, like RMI and the JSR 166 concurrency libraries. (Section 4.)

- We present a performance evaluation of our implementation on a variety of applications, including uses of futures and suspensions, and checking for transparency

Specification	Introduction	Coercion
Policy	Expression forms that introduce proxies	Expression forms that must operate on non-proxies
Implementation	Code to create a proxy (if any)	Code to implement run-time coercions (if any)

Table 1: Input Specification

violations (Section 5). Analysis times are comparable to those of similar static analyses. For futures and suspensions, the overhead of inserted dynamic coercions is negligible, while the performance benefits can be substantial. The analysis was also able to discover 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 (1) application and library classfiles to analyze, and (2) 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. These specifications are summarized in Table 1.

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 throughout the program. 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 3.7 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. So far we have not found such manual optimizations necessary.

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* [16]. As with our framework, 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. However, 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, where if e's type has qualifier Q, then proxy $\leq Q$. The effect of this expression is to coerce e to be a nonproxy. As an optimization, in the case that e is a local variable x, then x is treated flow-sensitively by the analysis: uses of x from then on treat it as being a nonproxy. During qualifier inference, expression forms in the user's coercion *pspec* could cause dynamic coercions to be inserted. These coercions are implemented following the user's *ispec*. For example, for wrapper proxies, a dynamic coercion is implemented by code that extracts x's underlying object. Moreover, to justify the flow-sensitivity of the analysis, code for a coercion 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 3.7.

3 Formal Development

This section describes our analysis formally and proves it sound. We model the analysis as an extension to Featherweight Java (FJ) [25], 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 $\mathcal{S}$ and I. The $\mathcal{S}$ constraints capture how proxies flow through the program, and the I constraints indicate where coercions may need to 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. Valid solutions σ and Ł to the constraints are denoted $\sigma \models S$ and $\sigma, L \models I$. 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 σ, L 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 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 and proofs 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 are annotated with a unique label l, used to designate where coercions should be inserted following inference. Note that there is no explicit coercion expression; these are only present in the target language FJ_Q.

Terms:

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

Types:

$$
\begin{array}{rcl}
Q & ::= & \mathsf{proxy} \mid \mathsf{nonproxy} \mid \kappa \\
\varphi & ::= & \{C_1, \ldots C_n\} \mid \alpha \\
N & ::= & \varphi^C \\
T & ::= & Q\, N
\end{array}
$$

Figure 1: Syntax of FJ^i_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 $\overline{T}\ \overline{f}$, a constructor K, and a list of methods $\overline{M}$. Constructors K merely assign their arguments to fields, either directly or by invoking the superclass constructor. Method bodies consist of a single expression. We write $\overline{x}$ as shorthand for $x_1, \ldots, x_n$ (similarly for $\overline{C}, \overline{f}$, etc.) and write $\overline{M}$ for $M_1 \ldots M_n$ (no commas). We abbreviate operations on pairs of sequences similarly, writing $\overline{T}\ \overline{f}$ for $T_1\ f_1, \ldots, T_n\ f_n$, where n is the length of $\overline{T}$ and $\overline{f}$. 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**.

Expressions e are as in FJ, with a few extensions. The syntax **makeproxy** e designates where a proxy should be introduced. Our formalism treats proxies generically,

without considering how particular proxies might be implemented. In particular, the operational semantics merely "tags" the result of evaluating e as being a possible proxy. We also have support for local variables (**let**s) and **if then else** expressions to illustrate effects of flow sensitivity, described below.

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 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, assume 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 set types N, rather than types T—no qualifier is necessary because it is assumed to be nonproxy.

The proxy analysis should take a normal Java program and infer 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, and is further justified below. Our implementation handles the full Java language.

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. Moreover, we include a well-formedness condition, 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 $\mathsf{proxy} \leq \mathsf{nonproxy}$ then $\mathsf{nonproxy}\ \{B\}^B \leq \mathsf{proxy}\ \{B,C\}^A$. 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.

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

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

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

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

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

$$\text{SubTyp}\ \frac{Q \leq Q' \qquad N \leq N'}{Q\,N \leq Q'\,N'}$$

Figure 2: Subtyping Rules

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 when 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, which expressions 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 e's label l is included in a set L. The output type of this judgment always has a nonproxy qualifier; this will be witnessed by inserting coercions during transformation. The (I-CoerceVar) rule is similar, except that the variable x in the input environment is rebound in the output environment to its coerced type. This flow-sensitive treatment allows expressions in the continuation to recognize that the variable has already been coerced, and need not be coerced again.

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

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

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

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

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

$$\text{MType-N}\ \frac{\mathit{mtype}(C_i,m) = \overline{T_i} \to U_i \quad \text{for all } i}{\mathit{mtype}(\{C_1,\ldots C_n\}^{C_0}) = \overline{T_1} \to U_1, \ldots \overline{T_n} \to U_n}$$

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

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

$$\text{Override}\ \frac{\begin{array}{c}\mathit{strip}(\mathit{mtype}(m,D)) = (\{\ldots\}^{D_1} \ldots \{\ldots\}^{D_n}) \to \{\ldots\}^{D_0} \\ \mathit{strip}(\overline{T} \to U) = (\{\ldots\}^{C_1} \ldots \{\ldots\}^{C_n}) \to \{\ldots\}^{C_0} \\ \overline{C} = \overline{D} \qquad\qquad C_0 = C_0\end{array}}{\mathit{override}(m, D, \overline{T} \to U)}$$

Figure 3: FJ_Q and FJ_Q^i: Auxiliary Definitions

Most inference rules thread the output context of one subexpression to the input context of another. When typing $\Gamma \vdash \overline{e} : \overline{T}; \Gamma'$, 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 type-

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

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

$$\text{I-If}\ \frac{\begin{array}{c}\Gamma \vdash_c e_1 : \mathsf{nonproxy}\ N; \Gamma_1 \\ \Gamma_1 \vdash_i e_2 : T_2; \Gamma_2 \qquad \Gamma_1 \vdash_i e_3 : T_3; \Gamma_3 \\ T_2 \leq T \qquad T_3 \leq T \qquad T' = \mathit{embed}(\mathit{strip}(T)) \\ \Gamma' = \mathit{merge}(\Gamma_2, \Gamma_3)\end{array}}{\Gamma \vdash_i (\textbf{if}\ e_1\ \textbf{instanceof}\ N\ \textbf{then}\ e_2\ \textbf{else}\ e_3)^l : T'; \Gamma'}$$

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

$$\text{I-Invoke}\ \frac{\begin{array}{c}\Gamma \vdash_c e_0 : \mathsf{nonproxy}\ N; \Gamma' \qquad \Gamma' \vdash_i \overline{e} : \overline{S}; \Gamma'' \\ \mathit{mtype}(m, N) = \overline{T_1} \to Q_1\ \varphi_1^C, \ldots \overline{T_n} \to Q_n\ \varphi_n^C \\ \overline{S} \leq \overline{T_i} \qquad Q_i\ \varphi_i^C \leq \kappa\ \alpha^C \qquad \text{for all } i \qquad \kappa, \alpha\ \text{fresh}\end{array}}{\Gamma \vdash_i (e_0.m(\overline{e}))^l : \kappa\ \alpha^C; \Gamma''}$$

$$\text{I-New}\ \frac{\mathit{fields}(\{C\}^C) = \overline{T}\ \overline{f} \qquad \Gamma \vdash_i \overline{e} : \overline{S}; \Gamma' \qquad \overline{S} \leq \overline{T}}{\Gamma \vdash_i (\textbf{new}\ C(\overline{e}))^l : \mathsf{nonproxy}\ \{C\}^C; \Gamma'}$$

$$\text{I-Cast}\ \frac{\begin{array}{c}\Gamma \vdash_c e : \mathsf{nonproxy}\ \varphi'^D; \Gamma' \\ \varphi' \subseteq \mathit{subtypes}(D) \\ \alpha \subseteq (\mathit{subtypes}(C) \cap \varphi') \qquad \alpha\ \text{fresh}\end{array}}{\Gamma \vdash_i ((\alpha^C)e)^l : \mathsf{nonproxy}\ \alpha^C; \Gamma'}$$

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

$$\text{I-CoerceExp}\ \frac{\begin{array}{c}\Gamma \vdash_i \mathcal{E}^{l_0} : Q\ N; \Gamma' \qquad \mathcal{E} \neq x \qquad l_0\ \text{fresh} \\ \mathsf{proxy} \leq Q \Rightarrow l \in L\end{array}}{\Gamma \vdash_c \mathcal{E}^l : \mathsf{nonproxy}\ N; \Gamma'}$$

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

Figure 4: FJ^i_Q: Inference for Expressions

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

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

Figure 5: FJ^i_Q: Inference for Classes and Methods

checking of the body is completed, the x binding is removed from output context Γ' when returned as the result of the entire let expression.

- 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 $\mathit{merge}(\Gamma_1,\Gamma_2)$ is the environment Γ' such that for each x in $\mathit{dom}(\Gamma_1)\cap\mathit{dom}(\Gamma_2)$, $\Gamma'(x)=T$ where $\Gamma_1(x)\leq T$ and $\Gamma_2(x)\leq T$.

- The (I-Invoke) rule uses the auxiliary function $\mathit{mtype}(m,N)$ to extract the types of the individual methods of the classes in the set of N (*mtype* and other auxiliary functions are shown in Figure 3). Subtyping constraints between the arguments passed to the method are only imposed on these methods, rather than all methods that are the subtypes of the bound, as would be the case in FJ. 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 places two conditions on a cast expression. First, the target settype α^C must only contain class names that are subtypes of the bound C. The predicate $\mathit{subtypes}(C)$ is the set of all subtypes of C defined in the class table CT. In addition, these types should be intersected with the current type of e, that is, φ'. There are three possible outcomes. First, if D is a supertype of C, 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 D is a subtype of C, then the intersection will have no

effect, since all the class names in α, 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 solution of α will be empty, signaling that we have type error.

- The (I-MakeProxy) rules requires that e in **makeproxy** e be a nonproxy. 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.

Finally, we use auxiliary function $\mathit{embed}(s)$ to annotate a Java (FJ) type s with fresh qualifier and set-type variables as described above. We define $\mathit{strip}(T)$ as the converse.

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]. We are planning to do this as part of future work.

3.4 Constraint Solving

$$
\begin{array}{lcl}
\mathcal{S} & \cup & \{Q\ \varphi^C \leq Q'\ \varphi'^D\} \Rightarrow \\
\mathcal{S} & \cup & \{Q \leq Q'\} \cup \{\varphi \subseteq \varphi'\} \cup \\
 & & \{\varphi \subseteq \mathit{subtypes}(C)\} \cup \{\varphi' \subseteq \mathit{subtypes}(D)\} \cup \\
 & & \{C \leq D\} \\
 & & \\
\mathcal{S} & \cup & \{\alpha \subseteq (\mathit{subtypes}(C) \cap \varphi)\} \Rightarrow \\
\mathcal{S} & \cup & \{\alpha \subseteq \mathit{subtypes}(C)\} \cup \{\alpha \subseteq \varphi\}
\end{array}
$$

Figure 6: Subtype Constraint Reduction

Proxy inference generates subtyping/subset constraints of the forms $T \leq U$ and $\alpha \subseteq (\mathit{subtypes}(C) \cap \varphi)$, and implication constraints of the form $\mathsf{proxy} \leq Q \Rightarrow l \in L$. Call the set of subtyping constraints $\mathcal{S}$, and the set of implication constraints I. We can solve these constraints as follows.

We can reduce $\mathcal{S}$ by continuously applying the rewriting rules shown in Figure 6. When finished, all constraints will have the following forms: $C \leq D$, $\varphi \subseteq \varphi'$, and $Q \leq Q'$. The first form consists of subtyping requirements determined by the program; if these do not hold then the program would not have typechecked using the FJ type system.

The remaining two forms can be solved by standard techniques. In particular, the qualifier constraints in $\mathcal{S}$ 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 [38]. The set-type constraints in $\mathcal{S}$ are subset constraints, as occur Andersen-style points-to analysis. Given n such constraints, these can be solved in $O(n^3)$ time [2].

Note that our rules are set up to favor a *least* solution to the qualifier constraints, which effectively delays a coercion until absolutely necessary.

A solution σ to constraints in $\mathcal{S}$ 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 $Q_1 \leq Q_2 \in \mathcal{S}$ we have $\sigma(Q_1) \leq \sigma(Q_2)$, and similarly for set-type constraints. We write $\sigma \models \mathcal{S}$ if σ is a solution of $\mathcal{S}$.

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

3.5 Transformation

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

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

$$
\begin{array}{lcl}
\mathcal{T}[\![e]\!] & \Rightarrow & coerce(e) \\
coerce(\mathcal{E}^l) & \equiv & \begin{cases} \textbf{coerce}\ \mathcal{T}[\![\mathcal{E}]\!] & l \in L \\ \mathcal{T}[\![\mathcal{E}]\!] & otherwise \end{cases}
\end{array}
$$

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

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 (this 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, 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 signal a possible violation. Rather than insert a coercion, 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 Figures 9 and 10 in the Appendix. Typechecking FJ_Q is straightforward, and similar to inference on FJ^i_Q.

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'). The complete transition rules are presented in the Appendix (Figure 12).

Since this is a qualified system, the store maps variables to *qualified store values*, which are store values h paired with a run-time representation of its 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. All qualified store values that are used concretely must have qualifier nonproxy, indicating that the actual value is available.

We can now state the *progress* and *preservation* lemmas for FJ_Q:

Lemma 3.1 (Progress) *Given that* $\Gamma \vdash (S,e) : T; \Gamma'$*, 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* $\Gamma \vdash (S,e) : T; \Gamma'$*, and that* $(S,e) \rightarrow (S',e')$*, then* $\Gamma' \vdash (S',e') : U; \Gamma''$ *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* $\Gamma \vdash e : T; \Gamma'$*, then either*

- $(\emptyset,e) \rightarrow^* (S,x)$ *for some S and x.*
- $(\emptyset,e) \rightarrow^* (S,e')$ *for some S and* e'*, such that* (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. The proofs of progress and preservation are by induction on the typing derivation, 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 class table* CT *for which* $\vdash_i CT$,
- *A substitution* σ *and label set* L,
- *Either* $\Gamma \vdash_i \mathcal{E}^l : T; \Gamma'$, *s.t.* $l \notin L$, *or* $\Gamma \vdash_c \mathcal{E}^l : T; \Gamma'$, *generating constraints* $\mathcal{S}$ *and* I.
- $\sigma \models \mathcal{S}$ *and* $\sigma, L \models I$.
- $\mathcal{T}[\![\mathcal{E}^l]\!] \Rightarrow e$, *and* $\mathcal{T}[\![CT]\!] \Rightarrow CT'$.

Then $\vdash CT'$, *and using class table* CT', $\sigma(\Gamma) \vdash e : \sigma(T); \sigma(\Gamma')$

The proof is by induction on the inference derivation. All proofs can be found in the Appendix.

Notes Compared to past work in flow-sensitive type qualifiers, flow-sensitivity in our system is significantly simpler. The approach of Foster et al. [17] 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 [16, 17], for example), it would not change the character of our approach, adding only unnecessary complication.

3.7 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. For example, Shankar et al. [41] 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.

Another application would 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 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, and how we used it to implement asynchronous method invocations in Java.

4.1 Framework Implementation

Our analysis is implemented as a bytecode-to-bytecode transformation. The static analysis is based on the Spark framework [31] for implementing points-to analysis for Java, both for the task of tracking proxies and for generating the set types based on points-to information. Spark uses Soot [44] to transform class files into an intermediate representation, similar to SSA, called Jimple. 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 incorporating multiple nodes, one per use of a possible proxy.

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

1. The `AsyncGen` class implements the introduction portion of the *pspec*, defining syntactic patterns that indicate where proxies are introduced. Introduction patterns must, of course, be legal Java (or more precisely Jimple) syntax that could have been compiled to bytecode.

2. The `Policy` class implements the coercion portion of the *pspec* using a visitor to specify which expressions require non-proxies.

3. The `ClaimTransformer` class implements the *ispec*, defining how call sites that create proxies are rewritten, and how coercions are implemented. It may direct that additional classes, like the `ProxyHandle` class described below, should be linked into the transformed application.

Because Jimple represents typed bytecode, coercions that assign back to the original variable must be well-typed. Therefore, for each Jimple variable `x` that could be a proxy, having program type *A*, we give it type `Object` instead. Whenever `x` is coerced, we assign the result to a newly-introduced variable `y` with type *A*, and replace all subsequent occurrences of `x` in the continuation with `y`. 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

```
result = Async.invoke(t,o.m(e1,e2,...));
```

The *pspec* indicates that the above syntactic form means that invoking method *m* on object *o* should be asynchronous, and that the result (if any) returned to the caller will be a future. The method's arguments $e_1, e_2, \ldots, e_n$ are evaluated in the current thread, and the asynchronous invocation is handled by a *thread manager t* (described below).

The *ispec* specifies that the implementation of an asynchronous call is as follows. First, the program creates a new object to encapsulate the method invocation on *m*. This object is a subclass of `ProxyHandle`, having the following signature:

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

The `Wrapper` interface simply defines a single `get` method:

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

The `ProxyHandle`'s `run` method, when invoked, will execute the method invocation $o.m(e_1,e_2,...)$, storing its result in a private field. Next, the `ProxyHandle` object is passed to the given thread manager t, which implements the JSR 166 `Executor` interface:

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

The thread manager will call the `ProxyHandle`'s `run` method to invoke the stored method. Finally, the `ProxyHandle` is returned to the caller (as `result`).

If some variable `x`, whose type in the program is `A`, could contain a future, and the analysis signals it must be coerced, the *ispec* implements the coercion with the following check

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

The call to `get` will extract the result, synchronizing and `wait` if the result is not yet available. The result of the coercion is treated flow-sensitively as described above.

Any implementation of `Executor` can be used as the thread manager. For example, the JSR 166 `ThreadPoolExecutor` class provides an extensible thread pool implementation. We have used variants of this class, as well as our own `ForkExecutor` and `BoundedForkExecutor`, which create new threads for each invocation, and a `ThreadPerObjectExecutor`, which keeps a map of objects to executors, delegating an invocation to its object's personal executor, as with *active objects* [30]. We implement lazy method invocation by using a `LazyExecutor` that simply stores the captured invocation in the `Wrapper`, and then implements `get` (called when the wrapper is claimed), to perform the invocation and return the result.

Note that programmers can influence where claims occur by performing "null" casts. That is, the expression $(N)e$ 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 very easily specify that a method should be asynchronous, or undo a previously asynchronous invocation. In contrast, doing this work by hand can require extensive code changes, either requiring an adapter class or per-call site modifications; we show an example in Section 5.2. Programmers also have control over how the invocation is handled, by specifying a thread manager.

4.3 Fine-grained Parallelism

Because the cost of thread creation and synchronization can be high, asynchronous method invocations should be used judiciously. In particular, asynchronous calls will likely improve performance only for potentially blocking or long-running operations, like remote method invocations, accesses to a database or the file system, etc. Our framework could handle more fine-grained parallelism given runtime support for faster and/or lazy thread creation, as discussed in Section 6.

4.4 Exceptions

If an asynchronous method call throws an exception E, that exception is stored in the call's future. We the future is claimed, the exception E is re-thrown.[2] This presents some challenges to the analysis.

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, and we extend method declarations 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 10 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:

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

[2] This differs from a JSR 166 `Future`, 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.

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 `ProxyHandle` 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 avoid creating a value having qualifier proxy$E$, by ensuring that expressions e in **makeproxy** e never throw exceptions. This is done 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. The user can also specify a default value for the object to be returned by a claim. Our experience is that this approach works quite often.

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. Many applications we have considered result in no errors because an unclaimed proxy will 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, such as `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 superset of the handlers mentioned in the proxy. We plan to experiment with this idea to see how useful it is in practice.

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.

4.5 Synchronization

Concurrent programs must balance safety and liveness, by guarding against race conditions and invariant violations, and preventing deadlock. When using asynchronous

test	tot (s)	per-check (ns)	% ovr
no claim	2.154	*n/a*	*n/a*
spurious claim	2.401	35	10%
necessary claim	3.567	141	65%

Table 2: Overhead of inserted claims, $N = 10^7$

method calls, programmers must use synchronization, immutability, and other techniques to achieve these goals—no automatic support is provided. This makes our approach no worse, and no better, than standard Java thread programming.

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, as is done in some past work [30, 7, 26, 18], but rejected this idea because of its 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 [33, 29, 8] and language-level transactions [21] aim to realize this goal. By not making any assumptions about synchronization, we can readily incorporate good results from these projects.

5 Evaluation

We evaluate our framework in terms of (1) analysis effectiveness (how does it impact the run-time of the instrumented program), (2) analysis performance (how fast is the analysis), and (3) programming benefit (how does our framework simplify the programming task). We present a number of applications of both wrapper proxies and transparency checking to show the costs and benefits of our approach.

5.1 Claim Overhead

Wrapper proxies can flow to potentially many parts of the program, causing our analysis to instrument classes with unnecessary claims. To measure the performance overhead of necessary as well as spurious 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, and 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, so will never be rewritten to be the enclosed object). The results are shown in Table 2 for $N = 10^7$ (other values of N showed a similar relationship).

Spurious claims consist of essentially two instructions at runtime: an `instanceof` check and a cast. Our measurements show this to add 10% to the loop running time. Necessary claims require an additional 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 spurious claims from then on. The overhead is artificially bad because in actual applications (1) all method calls will not require claims, and (2) method calls will perform real work, dwarfing the cost of claims to program running time.

5.2 Non-blocking Work Service

A simple application of futures is converting a blocking service into a non-blocking service, as suggested by the following example taken from the JSR 166 API documentation [12]. Here we go through this example to show how our framework greatly simplifies the programming process.

Consider the following interface:

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

Using JSR 166 `Futures`, a nonblocking variant of this interface can define the `serve` method to return `Future<Response>` instead. The programmer must then build an adapter class to wrap the blocking service, as shown in Figure 8.

Old clients of `BlockingService` objects must rewrite their code, first to wrap the original object with the adapter, then to claim the `Response` objects from their corresponding `Future<Response>` wrappers. Futures should be claimed at the last possible program point before their values are needed. Clients of the new non-blocking service must therefore sprinkle claims into their code directly before the `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. Moreover, the programmer must decide when a now-futurized `Response` object is passed to a method, whether to modify the method to accept a future as a parameter, or claim the future before invoking the method. The first option can be difficult or impossible if futures flow into library routines or third-party components, while the second option is unattractive because the client may pass the `Response` into a method that performs a large amount of work before touching the `Response` object.

This is a fair amount of programming overhead for a simple conceptual change. Moreover, a similar overhead is required to undo the change. Using our framework, we can achieve the same results in the best case by simply changing existing method calls

```
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

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

to be

```
Async.invoke(executor,adapter.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 4.4.

We wrote a simple implementation of `BlockingService` whose `serve` method extracts information from a database. Performance measurements for the analysis are shown in Table 3. These were performed on a 2 GHz dual-processor Xeon with 2 GB RAM, running RedHat Linux 8.0 (Linux kernel version 2.4.20). Here we show the results of both our flow-sensitive analysis (FS), and a flow-insensitive variant of it (FI). The results show the benefit of flow-sensitivity to precision: fewer classes are polluted with potential futures.

Analysis	Time	classes			
		analyzed	w/ fut.	re-written	claims
FI	88	1010	25	1	3
FS	131	1010	16	1	2
spark	96	1010	*n/a*	*n/a*	*n/a*

Table 3: Proxy Analysis Performance on BlockingService

We experimented with a field-sensitive variant of our analysis, but treating fields differently per allocation site is useless unless instance methods are also treated context-sensitively, taking into account the receiver that called the method. Since our algorithm is not context-sensitive, and since it is rare for one method to directly access the fields of another object, field-sensitivity never showed a benefit for our applications, and so we do not consider it further.

The flow-sensitive analysis takes somewhat more time to run than the flow-insensitive version, since 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. Our implementation currently incorporates nodes for flow-sensitive qualifiers into the same graph used by the points-to analysis for set types. This causes unnecessary propagation during constraint solving. Therefore, we could reduce the total running time by separating the two graphs. In this way, we would process fewer total classes with regard to qualifiers (16 rather than 25 in this case), and reduce the total time of the analysis. We plan to make this change straightaway.

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 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 [32], and recent work has considered the idea for Java [37, 43].

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.

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

Analysis	Time	classes			
		analyzed	w/ fut.	re-written	claims
FI	139	1319	17	3	3
FS	218	1319	9	2	1
spark	126	1320	*n/a*	*n/a*	*n/a*

Table 4: Proxy Analysis Performance on Asynchronous RMI benchmark

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

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

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 in the case that the client application wishes to invoke `findService` n times to create a composed service, because each call will wait until the prior service is found. Therefore, 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, serviceName))`. The analysis recognizes this syntax as introducing a wrapper proxy, and 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. The analysis times for this benchmark are shown in Table 4.

We ran some 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 are shown in Table 5, with all times in milliseconds.

We can see that for local area traffic, the added parallelism nets little performance gain, due to the rapid round trip times, as compared to the cost of thread management. However, 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

Version	Services requested and used									
	1	2	3	4	5	6	7	8	9	10
Orig	11	22	30	41	54	60	85	78	96	104
Async	11	24	32	43	53	61	76	81	90	101
Orig + delay	100	192	282	370	462	562	647	738	828	914
Async + delay	100	107	110	120	124	137	138	143	151	156

Table 5: Elapsed time (s) of Peer-to-Peer RMI application with varying workload

this cost.

Of course, these results could have been achieved by rewriting the application by hand to capture the invocation, and 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 of through the use of interface proxies. In this application, 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 following policy and implementation specification:

Policy Calls to `Proxy.newProxyInstance(...)` introduce proxies. All expressions that are identity-revealing must operate on non-proxies. 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), and any requirement of a coercion signals a possible transparency violation, since it suggests that a proxy is used an identity-revealing context. Therefore, the analysis signals that the coercion point is a potential transparency violation.

We ran our checker on two examples: an XML-based implementation of SOAP over RMI that uses dynamic proxy classes [42], and the Soot bytecode analysis framework [44]. In the former case, we simply instructed the analysis to track 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 would simulate a user wishing to proxy an object returned by one of these methods, e.g., to perform profiling, but wanting to ensure that transparency will not be violated.

Version	Time (s)		# of classes		Errors
	FI	FS	analyzed	with proxies	
Orig	227	289	2087	3	0/0
Error	226	300	2087	4	1/1
Spark	230	*n/a*	2087	*n/a*	*n/a*

Table 6: Analysis Performance for SOAP/RMI (Dynamic Proxy) Example

Example	Time (s)		# of classes		Errors
	FI	FS	analyzed	with proxies	
1	366	510	2510	24	0
2	368	522	2510	24	7
3	361	526	2510	1	1
spark	354	*n/a*	2510	*n/a*	*n/a*

Table 7: Analysis Performance for 3 Soot Examples

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. Then we inserted a single violation and re-ran the test, which discovered the violation. Table 6 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-insensitive 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.

The Soot examples for the three different methods are shown in Table 7. We looked at the reported violations, and verified in all cases that at least one was a genuine transparency violation that could lead to potential bug, though we did not verify them all. Once again, it was interesting to see that flow-sensitivity added no precision (only overhead!), and that our flow-insensitive analysis added little overhead to the original Spark analysis.

6 Related Work

6.1 Proxies

Gamma et al. [19] present many uses of the proxy design pattern, including remote references, lazy evaluation, and access control. Other uses[4] 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

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

been used in a variety of applications [42, 4].

6.2 Static Analysis

Our analysis draws upon techniques developed in other static analyses, including constraint-based analysis [1] and points-to analysis [10]. A points-to analysis tracks how aliases to a particular allocated data object are propagated throughout the program via assignments, dereferences, method calls, and so on. Our analysis uses the same constraint generation and solving machinery (in particular borrowing the framework of Spark [31]), but instead tracks how proxies flow throughout the program. This is similar Foster et al.'s qualifier inference [16], and propagation of changed types in CLA [22]. Indeed, we frame our analysis as a qualifier inference problem, and extend qualifier inference 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 [17], but their approach allows heap locations, and not just variables, to be treated flow-sensitively. This adds expressive power but significant complication.

Most work on points-to analysis has been for C [23]. For Java, most points-to analyses build on Andersen's analysis for C [2], which is flow-insensitive and context-insensitive [39, 31]. Recent work in points-to analysis has explored efficient context-sensitive versions of this analysis [14, 35], as well techniques for improving efficiency overall, such as by using on-demand constraint resolution rather than solving all the constraints in advance [22, 14, 20]. Allowing the analysis to be incremental, so that only changed classes are considered (and any ones on which they depend), can also improve analysis times [22, 40]. We intend to explore how these techniques could apply to our approach.

However, direct application may not be straightforward. In particular, our analysis is different than most applications of points-to-style analyses, which tend to be concerned with program optimization or safety checking. Our program transformation for wrapper proxies, consisting of rewriting of program types and introducing dynamic checks, adds new functionality. As a result, increasing the precision of the analysis using context-sensitivity would permit finer tracking of the flow of wrappers, particularly into container classes, but any user of a modified container class will be penalized by any added dynamic checks, even if a particular container instance will never contain wrappers. This could be avoided by "splitting" the class into a checked and non-checked version, but doing so would be difficult, since shared static data and proper typing must be preserved.

6.3 Asynchronous Method Calls and Futures

The notion of a future was popularized by Halstead in MultiLisp [27]. The syntax `(future` *e*`)` designates that expression *e* could be evaluated in a separate thread. The result of the expression is a *future* object. As MultiLisp is dynamically typed, the interpreter checks whether a value is a future when it is used concretely, and if so it extracts the actual value, or waits until it is available. This is called *touching* the future. Because

futures are known to the runtime system and hidden from the application, their use is transparent to the program. MultiLisp's garbage collector safely replaced futures with their actual values once available; our flow-sensitive analysis approximates this behavior. Flanagan and Felleisen [15] developed a whole-program static analysis for reducing the number of touches required; our analysis conversely adds needed touches based on the possibly flow of futures.

Liskov and Shrira developed a notion of a future for statically-typed languages, called a *promise* [32]. A promise, similar to a JSR 166 future for Java [28] mentioned in the introduction, is a type parameterized by the type of object it will ultimately compute. We found a number of applications of futures to statically-typed, object-oriented languages [34, 28, 24, 37, 11].

Halstead and others developed *lazy task creation* to dynamically adapt `future`-annotated programs to the runtime architecture and workload. Rather than create a new thread, each `future` expression is optimistically evaluated in the current thread. If the thread blocks or another processor becomes idle, then the caller's continuation is evaluated, establishing a future for the to-be-returned value. Alternatively, another processor can become idle, and steal the parent continuation. Similar techniques are employed by Cilk [18], and others. As future work, we are considering a similar scheme for Java.

A number of languages support so-called *active objects* [30], such as the SCOOP extension to Eiffel [7] and Io [26], which return futures. SCOOP inserts synchronization automatically, based on method preconditions. When a method is called, the invoking thread must wait until the conditions are satisfied before it can enter the object. Concurrent requests to enter the object are queued, and processed one at a time. While simple to use, programmers have less control. All concurrency occurs on a per-object basis, as opposed to per activity, which could severely limit performance without potentially unnatural program restructurings. Since many applications use concurrency for performance reasons (e.g., in multi-threaded server applications), imposing this restriction would be too onerous. Indeed, we allow programmers to indicate which thread manager they wish to use when executing a method asynchronously for exactly this reason.

Polyphonic C# [3] adds concurrency abstractions to C# based on the join calculus. Method declarations annotated as `async` are always invoked asynchronously. These methods never return results, hence there is no need for futures. Many aspects of their work are complementary with ours.

There has been some interest in developing asynchronous *remote* method invocation, to batch remote calls and thus amortize the delay of round-trip times. Promises were developed in this context. Raje et al. [37] propose an approach in which the returned future is made manifest to the programmer, adding to the programming burden. Sysala and Janecek [43] 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 flexible and easy-to-use framework for transparent programming with proxies in Java. Our framework recognizes this problem as one of qualifier inference, using improvements to qualifier inference algorithms to automatically introduce proxies, like futures, with a minimum of effort from the programmer, and ensure they are used transparently. We have formalized our framework and proven it sound. Our improvements to qualifier inference admitting new or improved applications. Using our framework, we have implemented a means for asynchronous and lazy method calls in Java, and have checked for possible transparency violations due to uses of the proxy design pattern. In the former case, applying asynchronous method calls to RMI nets significant performance gains with little programming effort, and in the latter case, a number of possible violations were detected.

Our analysis extends the Spark [31] points-to analysis, which is context-insensitive, and operates on the whole program. We are in the process of generalizing our framework to support arbitrary qualifiers, to support applications such as those mentioned in Section 3.7. In doing so, we plan to support more sophisticated context-sensitive analysis. We are also exploring how to make our analysis incremental, to avoid whole-program analysis when possible, easing software development. This should be a straightforward application of our explicitly-typed FJ_Q to all of Java, to allow reusing past results.

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

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 Cdric Fournet. Modern concurrency abstractions for C#. *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 *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 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 183–200. ACM Press, 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 *ICSE*, pages 442–452, 2002.

[9] Dynamic proxy classes. `http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html`. JDK 1.3 documentation.

[10] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI*, pages 242–256, 1994.

[11] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: spatial and temporal selection of reification. In *OOPSLA*, pages 27–46, 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 *OOPSLA*, pages 302–312, 2003.

[14] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In *PLDI*, pages 253–263, June 2000.

[15] Cormac Flanagan and Matthias Felleisen. The semantics of Future and its use in program optimizations. In *POPL*, pages 209–220, San Francisco, CA, January 1995.

[16] Jeffrey S. Foster, Manuel Fahndrich, and Alexander Aiken. A theory of type qualifiers. In *PLDI*, pages 192–203, May 1999.

[17] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-Sensitive Type Qualifiers. In *PLDI*, pages 1–12, Berlin, Germany, June 2002.

[18] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, pages 212–223, 1998.

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

[20] Samuel Z. Guyer and Calvin Lin. Client-driven pointer analysis. In *SAS*, June 2003.

[21] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA*, pages 388–402, 2003.

[22] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *PLDI*, pages 254–263, 2001.

[23] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE*, pages 54–61, New York, June 2001.

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

[25] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *TOPLAS*, 23(3):396–450, 2001.

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

[27] Robert H. Halstead Jr. Multilisp - a language for concurrent symbolic computation. *TOPLAS*, 7(4):501–538, 1985.

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

[29] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersen, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, 2001.

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

[31] Ondrej Lhotak and Laurie Hendren. Scaling Java points-to analysis using SPARK. *Lecture Notes in Computer Science*, 2622:153–169, 2003.

[32] Barbara Liskov and Liuba Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI*, pages 260–267, July 1988.

[33] Cristina Videira Lopes and Karl J. Lieberherr. Abstracting process-to-function relations in concurrency object-oriented applications. In *ECOOP*, volume 821 of *Lecture Notes in Computer Science*, pages 81–99, Bologna, Italy, July 1994. Springer.

[34] Dragos A. Manolescu. Workflow enactment with continuation and future objects. In *OOPSLA*, pages 40–51, 2002.

[35] Ana Milanova, Atanas Rountev, and Barbara Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. In *ISSTA*, pages 1–11, New York, July 2002.

[36] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 66–77, 1995.

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

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

[39] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for Java using annotated constraints. In *OOPSLA*, pages 43–55, 2001.

[40] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Fragment class analysis for testing of polymorphism in java software. In *ICSE*, pages 210–220, Piscataway, NJ, May 2003.

[41] 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.

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

[43] Tomás Sysala and Jan Janecek. Optimizing remote method invocation in Java. In *DEXA*, pages 29–35, September 2002.

[44] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot–a Java optimization framework. In *CASCON*, pages 125–135, 1999.

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, plus expressions **coerce** e. The typing rules for classes, methods, and expressions are shown in Figures 9 and 10. We have stripped labels from expressions for clarity, since they are not used. We extend typing to the abstract machine as described below.

The rules are basically straightforward analogues of the inference rules. Note that there are three rules for typing casts; these all come from FJ. The (UCast) rule types an upcast, the (DCast) rule types a dynamic downcast, and the (SCast) rule types a "stupid" cast. The last is a technical device 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 [36], 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 12.

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 $\mathbf{new}\,C(\overline{y})$, where the variables $\overline{y}$ index other qualified store values in S. Qualified store values are allocated by the following annotation rule, which replaces a store value h with a fresh variable x, and then maps that variable to h in the store S:

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

The other computation rules always operate on variables indexing the store, and so must "look up" the corresponding value for evaluation. For example, the following invocation 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.

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

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. This is illustrated in the premise of the invocation rule above.

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 coercion 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 such that it's 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 above coercion rule is well-defined for *all* qualified store values, not just those with qualifier proxy; this is critical because the subtyping rule $\mathsf{nonproxy} \leq \mathsf{proxy}$ 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 11. 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 indices 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.

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

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

Figure 9: FJ_Q: Typing Classes and Methods

$$\text{Var}\ \frac{}{\Gamma[x \mapsto T] \vdash x : T; \Gamma[x \mapsto T]}$$

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

$$\text{If}\ \frac{\Gamma \vdash e_1 : \mathsf{nonproxy}\ N_1; \Gamma_1 \quad \Gamma_1 \vdash e_2 : T_2; \Gamma_2 \quad \Gamma_1 \vdash e_3 : T_3; \Gamma_3 \quad T_2 \leq T \quad T_3 \leq T \quad \Gamma' = \mathit{merge}(\Gamma_2, \Gamma_3)}{\Gamma \vdash \textbf{if}\ e_1\ \textbf{instanceof}\ N\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : T; \Gamma'}$$

$$\text{Field}\ \frac{\Gamma \vdash e : \mathsf{nonproxy}\ N; \Gamma' \quad \mathit{fields}(N) = \overline{T}\ \overline{f}}{\Gamma \vdash e.f_i : T_i; \Gamma'}$$

$$\text{Invoke}\ \frac{\Gamma \vdash e_0 : \mathsf{nonproxy}\ N; \Gamma' \quad \Gamma' \vdash \overline{e} : \overline{S}; \Gamma'' \quad \mathit{mtype}(m, N) = \overline{T_1} \to U_1, \ldots \overline{T_n} \to U_n \quad \overline{S \leq T_i} \quad U_i \leq T \quad \text{for all } i}{\Gamma \vdash e_0.m(\overline{e}) : T; \Gamma''}$$

$$\text{New}\ \frac{\mathit{fields}(\{C\}^C) = \overline{T}\ \overline{f} \quad \Gamma \vdash \overline{e} : \overline{S}; \Gamma' \quad \overline{S} \leq \overline{T}}{\Gamma \vdash \textbf{new}\ C(\overline{e}) : \mathsf{nonproxy}\ \{C\}^C; \Gamma'}$$

$$\text{UCast}\ \frac{\Gamma \vdash e : \mathsf{nonproxy}\ M; \Gamma' \quad M \leq N}{\Gamma \vdash (N)e : \mathsf{nonproxy}\ N; \Gamma'}$$

$$\text{DCast}\ \frac{\Gamma \vdash e : \mathsf{nonproxy}\ M; \Gamma' \quad N \leq M \quad N \neq M}{\Gamma \vdash (N)e : \mathsf{nonproxy}\ N; \Gamma'}$$

$$\text{SCast}\ \frac{\Gamma \vdash e : \mathsf{nonproxy}\ N; \Gamma' \quad N \not\leq M \quad M \not\leq N \quad \textit{stupid warning}}{\Gamma \vdash (N)e : \mathsf{nonproxy}\ N; \Gamma'}$$

$$\text{MakeProxy}\ \frac{\Gamma \vdash e : \mathsf{nonproxy}\ N; \Gamma'}{\Gamma \vdash \textbf{makeproxy}\ e : \mathsf{proxy}\ N; \Gamma'}$$

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

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

Figure 10: FJ_Q: Expression Typing

$$\text{CheckNewQ}\ \frac{\mathit{fields}(\{C\}^C) = \overline{T}\ \overline{f} \qquad \Gamma(\overline{x}) = \overline{U} \qquad \overline{U} \leq \overline{T}}{\Gamma \vdash (Q, \mathbf{new}\, C(\overline{x})) : Q\ \{C\}^C}$$

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

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

Figure 11: FJ_Q: Program Typing

Transitions:

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

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

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

$$\text{TransCast}\ \frac{S(x) \;=\; (\mathsf{nonproxy}, \mathbf{new}\, D(\overline{y})) \qquad \{D\}^D \leq N}{(S, (N)x) \rightarrow (S, x)}$$

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

$$\text{TransIfTrue}\ \frac{S(x) \;=\; (\mathsf{nonproxy}, \mathbf{new}\, D(\overline{y})) \qquad \{D\}^D \leq N}{(S, \mathbf{if}\, x\, \mathbf{instanceof}\, N\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2) \rightarrow (S, e_1)}$$

$$\text{TransIfFalse}\ \frac{S(x) \;=\; (\mathsf{nonproxy}, \mathbf{new}\, D(\overline{y})) \qquad \{D\}^D \not\leq N}{(S, \mathbf{if}\, x\, \mathbf{instanceof}\, N\, \mathbf{then}\, e_1\, \mathbf{else}\, e_2) \rightarrow (S, e_2)}$$

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

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

Congruence rules:

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

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

Figure 12: FJ_Q: Operational Semantics

B Proofs

B.1 Progress

Lemma B.1 (Progress) *Given that* $\vdash (S, e_0) : T$*, then either* e_0 *is a variable* x*, or else* $(S, e_0) \rightarrow (S', e_0')$ *for some* S' *and* e_0'*,* unless the program reaches an erroneous state due to an impossible cast.

Proof The proof is by induction on the syntax of expressions that satisfy $\vdash (S, e_0) : T$ can take a step using a transition rule. For every non-value expression, we prove that it either can

- take a reduction step directly (base cases), or
- given that, if a sub-expression typechecks, then it can reduce, the whole expression can reduce (induction step).

Case analysis of the possible forms of expression e_0:

Case $e_0 \equiv (x)$: In this case the lemma is true by definition, the expression is a value.

Case $e_0 \equiv e.f$: From $\Gamma \vdash e_0 : T$ and [Field] we get $\Gamma \vdash e : \mathsf{nonproxy}\ N; \Gamma'$. Also $\mathit{fields}(C) = \overline{T}\ \overline{f}$.

- **Case** $e_0 \equiv x.f$: From $\vdash S : \Gamma$ and $\Gamma \vdash x : \mathsf{nonproxy}\ N; \Gamma'$ we deduce $S(x) = (\mathsf{nonproxy}, \mathbf{new}\ C(y))$ for a $\{C\}_C \leq N$. So, e_0 reduces by [TransField].
- **Case** $e_0 \equiv e.f$: From $\Gamma \vdash e : \mathsf{nonproxy}\ N; \Gamma'$ and the induction hypothesis, e_0 reduces by [C-CongruenceE].

Case $e_0 \equiv (e_1.m(\overline{e}))$: From $\Gamma \vdash e_0 : T$ and [Invoke] we get $\Gamma \vdash e_1 : \mathsf{nonproxy}\ N; \Gamma'$.

- **Case** $e_0 \equiv (x.m(\overline{y}))$: From $\Gamma \vdash x : \mathsf{nonproxy}\ N; \Gamma'$, and $\vdash S : \Gamma$ we have $S(x) = (\mathsf{nonproxy}, \mathbf{new}\ C(\overline{y}))$ for some $\{C\}_C \leq N$. Moreover by definition, from $\mathit{mtype}(m, C) = \overline{T} \rightarrow U$ we get that $\mathit{mbody}(m, C) = (\overline{z}, e_m)$. So, e_0 can reduce by [TransInvoke].
- **Case** $e_0 \equiv (e_1.m(\overline{e}))$: From $\Gamma \vdash e_1 : \mathsf{nonproxy}\ N; \Gamma'$ and the induction hypothesis, $e_1 \rightarrow e_1'$ and the whole expression reduces by [C-CongruenceE].
- **Case** $e_0 \equiv (x.m(\overline{e}))$: From $\Gamma \vdash e_0 : T$ and [Invoke] we get $\Gamma' \vdash \overline{e} : \overline{T}$, so by induction hypothesis, $\overline{e} \rightarrow \overline{e'}$ and the whole expression reduces by [C-CongruenceBarE].

Case $e_0 \equiv (\mathbf{new}\ C(\overline{e}))$:

- **Case** $e_0 \equiv (\mathbf{new}\ C(\overline{x}))$: Reduces by [TransAnnot].
- **Case** $e_0 \equiv (\mathbf{new}\ C(\overline{e}))$: From [New] and $\Gamma \vdash e_0 : T$ we get $\Gamma \vdash \overline{e} : \overline{S}; \Gamma'$. So, by the induction hypothesis e_0 can reduce by [C-CongruenceBarE].

Case $e_0 \equiv ((C)e)$:

- **Case** $e_0 \equiv ((N)x)$: From [DCast], [UCast] or [SCast] we have that $\Gamma \vdash x : \mathsf{nonproxy}\ M$, therefore from $\vdash S : \Gamma$ we have that $S(x) = (\mathsf{nonproxy}, \mathbf{new}\ D(\overline{x}))$, for a D that satisfies $\{D\}_D \leq M$. So, if $\{D\}_D \leq N$ then e_0 reduces by [TransCast], otherwise we have an erroneous stuck program.
- **Case** $e_0 \equiv ((C)e')$: From [DCast], [UCast] or [SCast] we have $\Gamma \vdash e' : T$, so by induction hypothesis, e_0 reduces by [C-CongruenceE].

Case $e_0 \equiv (\textbf{let}\ x = e_1\ \textbf{in}\ e_2)$:

Case $e_0 \equiv (\textbf{let}\ x = y\ \textbf{in}\ e_2)$: The term reduces by [TransLet].

Case $e_0 \equiv (\textbf{let}\ x = e_1\ \textbf{in}\ e_2)$: Given $\Gamma \vdash e_0 : T$ we have from [Let] $\Gamma \vdash e_1 : T'; \Gamma_1$. From induction hypothesis, we get that $e_1 \rightarrow e_1'$, therefore e_0 reduces by [C-CongruenceE].

Case $e_0 \equiv (\textbf{makeproxy}\ e')$:

Case $e_0 \equiv \textbf{makeproxy}\ x$: e_0 typechecks, so from [MakeProxy] we get $\Gamma \vdash x : \mathsf{nonproxy}\ N; \Gamma'$. From $\vdash S : \Gamma$ we get that $S(x) = (\mathsf{nonproxy}, \textbf{new}\ C(\overline{y}))$, for some C such that $\{C\}_C \leq N$. So, we can reduce by [TransProxy].

Case $e_0 \equiv \textbf{makeproxy}\ e'$: From $\Gamma \vdash e_0 : T$ and [MakeProxyCheck] we get that $\Gamma \vdash e' : T'$, therefore from the induction hypothesis, it reduces by [C-CongruenceE].

Case $e_0 \equiv (\textbf{coerce}\ e')$:

Case $e_0 \equiv (\textbf{coerce}\ x)$: From $\Gamma \vdash e_0 : T$ and [CoerceVarCheck] we get that Γ can be written as $\Gamma[x \mapsto Q\,N]$ such that: $\Gamma'[x \mapsto Q\,N] \vdash x : Q\,N; \Gamma'[x \mapsto Q\,N]$. From this and $\vdash S : \Gamma$ we get that S can be written as $S' \uplus \{x \mapsto (Q, \textbf{new}\ C(\overline{y}))\}$ for some C such that $\{C\}_C \leq N$. So, e_0 reduces by [TransCoerce].

Case $e_0 \equiv (\textbf{coerce}\ e')$: From $\Gamma \vdash e_0 : T$ and [CoerceExpCheck] we get that $\Gamma \vdash e' : Q\,N; \Gamma'$. Therefore, by induction hypothesis, e_0 reduces by [C-CongruenceE].

Case $e_0 \equiv (\textbf{if}\ e\ \textbf{instanceof}\ N\ \textbf{then}\ e\ \textbf{else}\ e)$

Case $e_0 \equiv (\textbf{if}\ x\ \textbf{instanceof}\ N\ \textbf{then}\ e_2\ \textbf{else}\ e_3)$ From $\Gamma \vdash e_0 : T$ and [If] we get that $\Gamma \vdash x : \mathsf{nonproxy}\ N_1$. From this and $\vdash S : \Gamma$ we get that $S(x) = (\mathsf{nonproxy}, \textbf{new}\ C(\overline{y}))$ where $\{C\}_C \leq N_1$. So, e_0 reduces by [TransIfTrue] or [TransIfFalse].

Case $e_0 \equiv (\textbf{if}\ e_1\ \textbf{instanceof}\ N\ \textbf{then}\ e_2\ \textbf{else}\ e_3)$ From $\Gamma \vdash e_0 : T$ and [If] we get that $\Gamma \vdash e_1 : \mathsf{nonproxy}\ N_1; \Gamma'$. Therefore, by induction hypothesis, e_0 reduces by [C-CongruenceE].

□

B.2 Preservation

Lemma B.2 (Preservation) *Given that* $\vdash (S, e_0) : T$, *and that* $(S, e_0) \rightarrow (S', e_0')$, *then* $\vdash (S', e_0') : U$ *such that* $U \leq T$.

Proof Induction proof:

- The lemma either holds for the program (S, e_0) directly, or
- if the lemma holds for all non-value sub-expressions of e_0, then it holds for (S, e_0).

Case analysis of the possible forms of program (S, e_0):

Case $(S, e_0) \equiv (S, x)$: In this case the program cannot take an evaluation step, therefore by definition the lemma is true.

Case $(S, e_0) \equiv (S, \underline{e.f})$: From $\Gamma \vdash e_0 : T; \Gamma'$ and [Field] we get $\Gamma \vdash e : \mathsf{nonproxy}\ N; \Gamma'$. Also $\mathit{fields}(C) = \overline{T}\ f$.

Case $e_0 \equiv x.f$: Then (S, e_0) reduces by [TransField]: $(S, x.f_i) \rightarrow (S, x_i)$. Given $\vdash (S, e_0)$, we have from [Field] that $\mathit{fields}(C) = \overline{T}\ \overline{f}$. By hypothesis $\vdash S : \Gamma$, which gives by [CheckState] that $\vdash S(x) : \Gamma(x)$. So from [New], $\Gamma \vdash x_i : S_i; \Gamma$ where $S_i \leq T_i$.

Case $e_0 \equiv e_1.f$: From the induction hypothesis, $(S, e_1) \rightarrow (S', e_1')$ and $\vdash (S, e_1) : T_1$ mean that $\vdash (S', e_1') : T_1'$ where $T_1' \leq T_1$. So, $\mathit{fields}(T_1) \subset \mathit{fields}T_1'$, and therefore, $\vdash (S', e_1'.f) : T$

Case $(S, e_0) \equiv (S, e_1.m(\overline{e}))$: From $\Gamma \vdash e_0 : T_0$ and [Invoke] we get $\Gamma \vdash e_1 : \mathsf{nonproxy}\ N; \Gamma'$.

Case $e_0 \equiv (o.m(\overline{y}))$: From [Invoke] we have:

$$\text{Invoke}\ \frac{\begin{array}{ccc} \Gamma \vdash o : \mathsf{nonproxy}\ N; \Gamma & & \Gamma \vdash \overline{y} : \overline{T_y}; \Gamma \\ \mathit{mtype}(m, N) = \overline{T_1} \rightarrow U_1, \ldots \overline{T_n} \rightarrow U_n & & \\ \overline{T_y} \leq \overline{T_i} \qquad U_i \leq V & & \text{for all } i \end{array}}{\Gamma \vdash e_0.m(\overline{e}) : V; \Gamma''}$$

For all the classes C_i that belong to the set N, $\mathit{mbody}(m, C_i) = (\overline{x}, e_i)$. Moreover, from $\Gamma \vdash o : \mathsf{nonproxy}\ N; \Gamma'$, and $\vdash S : \Gamma$ we have $S(o) = (\mathsf{nonproxy}, \mathbf{new}\, C(\overline{y}))$ for some $\{C\}_C \leq N$. Therefore, we get that $\mathit{mbody}(m, C) = (\overline{x}, e)$ for that C.

From [MBody-C] and [MBody-CSub], we get that for some ancestor D of $C \leq D$, we have $\mathit{mbody}(m, D) = (\overline{x}, e)$ and m is declared in $\overline{M}$ of D: $U\ m(\overline{T}\ \overline{x})\ \{\ \mathbf{return}\ e;\}$. Therefore, for D we have by [Method] that $\overline{x} : \overline{T}, \mathbf{this} : \mathsf{nonproxy}\ \{C\}_C \vdash e : T_e$ and $T_e \leq U$.

We know that (S, e_0) reduces by [TransInvoke] to $(S \uplus \{\overline{x} \mapsto S(\overline{y})\}, e[\mathbf{this} \mapsto o])$. Since $\overline{x} \notin \mathit{dom}(S)$ we can create a $\Gamma' = \Gamma \uplus \{\overline{x} \mapsto \overline{T}\} \uplus \{o \mapsto \mathsf{nonproxy}\ \{C\}_C\}$. Then, $S'(o) : \mathsf{nonproxy}\ \{C\}_C$ which means $S'(o) : \Gamma'(o)$. Furthermore, $\vdash S : \Gamma$, and the only new elements in S' are $\overline{x}$, for which $S(x_i) = S(y_i)$, and $S(y_i) : \Gamma(y_i)$. But, $\Gamma(\overline{y}) = \overline{T_y}$ and $\overline{T_y} \leq \overline{T}$, so, by [CheckStore], we have that $\vdash S' : \Gamma'$.

Finally, we also have that $\Gamma' \vdash e[\mathbf{this} \mapsto o] : T_e$ and $T_e \leq U$ from [Method], and $U \leq V$ from [Invoke]. Therefore $T_e \leq V$.

Case $e_0 \equiv (e_1.m(\overline{e}))$: $\vdash (S, e_0)$ and e_0 reduces by [C-CongruenceE], so by the induction hypothesis, $\vdash (S', e_0')$.

Case $e_0 \equiv (x.m(\overline{e}))$: $\vdash (S, e_0)$ and e_0 reduces by [C-CongruenceBarE], so by the induction hypothesis, $\vdash (S', e_0')$.

Case $(S, e_0) \equiv (S, \mathbf{new}\, C(\overline{e}))$:

Case $e_0 \equiv (\mathbf{new}\, C(\overline{y}))$: Reduces by [TransAnnot] to $(S \uplus \{x \mapsto (\mathsf{nonproxy}, \mathbf{new}\, C(\overline{y}))\}, x)$. From $\vdash (S, e_0) : T_0$ and [New] we have $\Gamma \vdash e_0 : \mathsf{nonproxy}\ \{C\}_C; \Gamma$. Moreover $S'(x) = (\mathsf{nonproxy}, \mathbf{new}\, C(\overline{y}))$. So, for $\Gamma' = \Gamma[x \mapsto \mathsf{nonproxy}\ \{C\}_C]$ we have: $S(z) : \Gamma(z)$ for every $z \in \mathit{dom}(\Gamma)$, $S'(x) : \Gamma'(x)$, and $\mathit{dom}(\Gamma') = \mathit{dom}(\Gamma) \cup \{x\}$. Therefore, $\vdash S' : \Gamma'$.

Also, $\Gamma' \vdash x : \mathsf{nonproxy}\ C$ and $\mathsf{nonproxy}\ \{C\}_C \leq T_0$.

Case $e_0 \equiv (\mathbf{new}\, C(\overline{e}))$: e_0 reduces by [C-CongruenceBarE], so by the induction hypothesis, $\vdash (S', e_0') : T'$ and $T' \leq T_0$.

Case $(S, e_0) \equiv (S, (N)e)$:

Case $e_0 \equiv ((N)x)$: In this case, the program either takes a step by [TransCast] or we have a stuck program due to a bad cast. If the program takes a step, it will reduce to $(S,(N)x) \rightarrow (S,x)$. The fact that the program takes a step means that $S(x) = (\mathsf{nonproxy}, \mathbf{new}\, D(\overline{y}))$ and $\{D\}_D \leq N$, by [TransCast]. So, x in the resulting program will have type $\mathsf{nonproxy}\ \{D\}_D$ where $\{D\}_D \leq N$.

Case $e_0 \equiv ((N)e)$: From [DCast], [UCast] or [SCast] we have $\Gamma \vdash e : T$, and e_0 reduces by [C-CongruenceE] to $(S',(N)e')$. So, by induction hypothesis, $\vdash (S',e') : T'$ where $T' \leq T$. Therefore e_0 will typecheck with [UCast], [DCast] or [SCast] with type $\mathsf{nonproxy}\ N$.

Case $(S,e_0) \equiv (S, \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2)$:

Case $e_0 \equiv (\mathbf{let}\ x = y\ \mathbf{in}\ e_2)$: Then (S,e_0) reduces by [TransLet] to $(S, \mathbf{let}\ x = y\ \mathbf{in}\ e_2) \rightarrow (S \uplus \{x \mapsto S(y)\}, e_2)$. Given that $\vdash (S,e_0) : T$, we know that $\Gamma \vdash e_0 : T$ and from [Let] we have that $\Gamma \vdash y : T_y$ and $\Gamma_1[x \mapsto T_y] \vdash e_2 : T$. So, for $\Gamma' = \Gamma_1[x \mapsto T_y]$, we have that $S'(z) : \Gamma'(z) \forall z \in dom(\Gamma)$ and that $S'(x) = S(y) : \Gamma(y)$. But $\Gamma'(x) = \Gamma(y)$, so $\vdash S' : \Gamma'$. Moreover, from [Let] we know that $\Gamma' \vdash e_2 : T$.

Case $e_0 \equiv (\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2)$: Given $\Gamma \vdash e_0 : T$ we have from [Let] $\Gamma \vdash e_1 : T_1; \Gamma_1$. From hypothesis, e_0 reduces (by [C-CongruenceE]) so $e_1 \rightarrow e_1'$. By the induction hypothesis, $\vdash (S',e_1') : T_1'$ and $T_1' \leq T_1$. So, $\vdash (S',e_0') : T'$.

Case $(S,e_0) \equiv (S, \mathbf{makeproxy}\ e')$:

Case $e_0 \equiv \mathbf{makeproxy}\ x$: e_0 typechecks, so from [MakeProxy] we get $\Gamma \vdash e_0 : \mathsf{proxy}\ N$ and $\Gamma \vdash x : \mathsf{nonproxy}\ N$

From $\vdash S : \Gamma$ we get that $S(x) = (\mathsf{nonproxy}, h)$, where $h = \mathbf{new}\, C(\overline{y})$ for some C such that $\{C\}_C \leq N$. Reduction by [TransProxy] gives $(S \uplus \{y \mapsto (\mathsf{proxy}, h)\}, y)$. For $\Gamma' = \Gamma[y \mapsto \mathsf{proxy}\ N]$, we have $\vdash S' : \Gamma'$. Also, $\Gamma' \vdash y : \mathsf{proxy}\ N$.

Case $e_0 \equiv \mathbf{makeproxy}\ e$: From $\Gamma \vdash e_0 : T$ and [MakeProxyCheck] we get that $\Gamma \vdash e : T_e$. Also, by hypothesis, it reduces by [C-CongruenceE] to $(S', e_0 \equiv \mathbf{makeproxy}\ e')$ where $\vdash S' : \Gamma'$, $\Gamma' \vdash e' : T_e'$ and $T_e' \leq T_e$. So, $\Gamma' \vdash \mathbf{makeproxy}\ e' : T'$ and $T' \leq T$.

Case $(S,e_0) \equiv (S, \mathbf{coerce}\ e)$:

Case $e_0 \equiv (\mathbf{coerce}\ x)$: From $\Gamma \vdash e_0 : T$ and [CoerceVarCheck] we get that Γ can be written as $\Gamma_1[x \mapsto Q\ N]$ such that: $\Gamma_1[x \mapsto Q\ N] \vdash x : Q\ N; \Gamma_1[x \mapsto Q\ N]$. From this and $\vdash S : \Gamma$ we get that S can be written as $S' \uplus \{x \mapsto (Q, \mathbf{new}\, C(\overline{y}))\}$ for some C such that $\{C\}_C \leq N$.

By hypothesis, e_0 reduces by [TransCoerce]: $(S \uplus \{x \mapsto (Q,h)\}, \mathbf{coerce}\ x) \rightarrow (S \uplus \{x \mapsto (\mathsf{nonproxy}, h)\}, x)$.

So, for $\Gamma' = \Gamma_1[x \mapsto \mathsf{nonproxy}\ N]$ we have $\vdash S' : \Gamma'$ and $\Gamma' \vdash x : \mathsf{nonproxy}\ N$, where $\mathsf{nonproxy}\ N \leq Q\ N$.

Case $e_0 \equiv (\mathbf{coerce}\ e)$: By hypothesis: $(S, \mathbf{coerce}\ e) \rightarrow (S', \mathbf{coerce}\ e')$ and $\vdash (S,e_0) : T$, which gives from [CoerceExpCheck] that $\vdash (S,e) : T_e$.

So, by induction hypothesis we get that $\vdash (S',e') : T_e'$ and $T_e' \leq T_e$. Therefore, $\vdash (S', \mathbf{coerce}\ e') : T'$ where $T' \leq T$.

Case $(S,e_0) \equiv (S, \mathbf{if}\ e\ \mathbf{instanceof}\ N\ \mathbf{then}\ e\ \mathbf{else}\ e)$

Case $e_0 \equiv (\textbf{if } x \textbf{ instanceof } N \textbf{ then } e_2 \textbf{ else } e_3)$ From $\Gamma \vdash e_0 : T$ and [If] we get that $\Gamma \vdash x : \mathsf{nonproxy}\ N_1$. From this and $\vdash S : \Gamma$ we get that $S(x) = (\mathsf{nonproxy}, \textbf{new } C(\overline{y}))$ where $\{C\}_C \leq N_1$. Also, (S, e_0) reduces by [TransIfTrue] or [TransIfFalse], to (S, e_2) or (S, e_3).

So, for $\Gamma' = \Gamma$, we have that $\vdash S' : \Gamma'$ since S did not change, and that $\Gamma' \vdash e_2 : T_2$ and $\Gamma' \vdash e_3 : T_3$, where from [If] we had that $T_2 \leq T$ and $T_3 \leq T$. Therefore, in either case $\Gamma' \vdash e_0' : T'$ and $T' \leq T$.

Case $e_0 \equiv (\textbf{if } e_1 \textbf{ instanceof } N \textbf{ then } e_2 \textbf{ else } e_3)$ From $\Gamma \vdash e_0 : T$ and [If] we get that $\Gamma \vdash e_1 : \mathsf{nonproxy}\ N_1; \Gamma'$. Also by hypothesis, e_0 reduces by [C-CongruenceE], so $(S, e_1) \rightarrow (S', e_1')$. By the induction hypothesis, there is a Γ' such that $\vdash S' : \Gamma'$ and $\Gamma' \vdash e_1' : T_1'$ where $T_1' \leq \mathsf{nonproxy}\ N_1$. It follows that $T_1' = \mathsf{nonproxy}\ N_1'$ and $N_1' \leq N_1$.

Therefore, $\Gamma' \vdash e_0' : T'$ and $T' \leq T$.

□

B.3 Inference Soundness

Theorem B.3 (Inference Soundness) *Given*

- *A class table CT,*
- *A substitution* σ *and label set L,*
- *Either* $\Gamma \vdash_i \mathcal{E}^l : T; \Gamma'$*, s.t.* $l \notin L$*, or* $\Gamma \vdash_c \mathcal{E}^l : T; \Gamma'$*, generating constraints* $\mathcal{S}$ *and I.*
- $\sigma \models \mathcal{S}$ *and* $\sigma, L \models I$*.*
- $\mathcal{T}[\![\mathcal{E}^l]\!] \Rightarrow e$*, and* $\mathcal{T}[\![CT]\!] \Rightarrow CT'$*.*

Then using class table CT'*,* $\sigma(\Gamma) \vdash e : \sigma(T); \sigma(\Gamma')$*.*

Proof

- The theorem either holds for $\mathcal{E}$ directly, or
- if the theorem holds for $\mathcal{E}_1, \ldots, \mathcal{E}_n$, then it holds for $\mathcal{E}$ whose subexpressions are $\mathcal{E}_1, \ldots, \mathcal{E}_n$

Given that the $\Gamma \vdash_i \mathcal{E}^l : T; \Gamma'$ or $\Gamma \vdash_c \mathcal{E}^l : T; \Gamma'$, the last step of the typing derivation will be one of:

Case [I-Var]: Then, $\mathcal{E} \equiv x$ and $\Gamma[x \mapsto T] \vdash_i x^l : T; \Gamma[x \mapsto T]$. The expression transformation is $\mathcal{T}[\![\S]\!] \Rightarrow x$. Then, $\sigma(\Gamma[x \mapsto T]) = \sigma(\Gamma)[x \mapsto \sigma(T)]$. If $\sigma T = T'$, then $\mathcal{T}[\![\S^l]\!]$ typechecks with [Var]: $\sigma(\Gamma)[x \mapsto T'] \vdash x : T'; \sigma(\Gamma)[x \mapsto T']$.

There are two cases for $\mathcal{T}[\![\mathcal{E}^l]\!]$.

- $l \notin L$: Then, $\mathcal{T}[\![\mathcal{E}^l]\!] \Rightarrow \mathcal{T}[\![\mathcal{E}]\!]$. Obviously, the theorem holds.
- $l \in L$: Then, $\mathcal{T}[\![\mathcal{E}^l]\!] \Rightarrow \textbf{coerce } \mathcal{T}[\![\mathcal{E}]\!] \Rightarrow \textbf{coerce } x$. From [CoerceVarCheck] we have that **coerce** x typechecks if $\Gamma_1[x \mapsto Q\,N] \vdash x : Q\,N; \Gamma_1[x \mapsto Q\,N]$. This is satisfied for $\Gamma_1 = \sigma(\Gamma)$ and $Q\,N = T'$.

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

Case [I-Let]: Then $\mathcal{E} \equiv \textbf{let}\, x = e_1\, \textbf{in}\, e_2$ and the transformed expression is $\mathcal{T}[\![\mathcal{E}]\!] = \textbf{let}\, x = \mathcal{T}[\![e_1]\!]\, \textbf{in}\, \mathcal{T}[\![e_2]\!]$ From [I-Let] we have that $\Gamma \vdash_i e_1 : T; \Gamma_1$ and $\Gamma_1[x \mapsto T] \vdash_i e_2 : T'; \Gamma'[x \mapsto T'']$. Therefore, by induction hypothesis, we get that $\sigma(\Gamma) \vdash \mathcal{T}[\![e_1]\!] : \sigma(T); \sigma(\Gamma_1)$ and $\sigma(\Gamma_1[x \mapsto T]) \vdash \mathcal{T}[\![e_2]\!] : \sigma(T'); \sigma(\Gamma'[x \mapsto T''])$. The latter can be written as: $\sigma(\Gamma_1)[x \mapsto \sigma(T)] \vdash \mathcal{T}[\![e_2]\!] : \sigma(T'); \sigma(\Gamma')[x \mapsto \sigma(T'')]$. Therefore, by [Let] we get that:

$$\sigma(\Gamma) \vdash \textbf{let}\, x = \mathcal{T}[\![e_1]\!]\, \textbf{in}\, \mathcal{T}[\![e_2]\!] : \sigma(T'); \sigma(\Gamma') \tag{1}$$

There are two cases for $\mathcal{T}[\![\mathcal{E}^l]\!]$:

- $l \notin L$: Then, $\mathcal{T}[\![\mathcal{E}^l]\!] \Rightarrow \mathcal{T}[\![\mathcal{E}]\!]$ and the theorem holds.
- $l \in L$: Then, $\mathcal{T}[\![\mathcal{E}^l]\!] \Rightarrow \textbf{coerce}\, \mathcal{T}[\![\mathcal{E}]\!] \Rightarrow \textbf{coerce}\, e$. In that case, from [CoerceExpCheck] and equation (1) we have that **coerce** e typechecks.

Therefore, in both cases, $\sigma(\Gamma) \vdash \mathcal{T}[\![\mathcal{E}^l]\!] : \sigma(T); \sigma(\Gamma')$.

Case [I-If]: Then

$$\mathcal{E} \equiv \textbf{if}\, e_1\, \textbf{instanceof}\, N\, \textbf{then}\, e_2\, \textbf{else}\, e_3$$

and the transformed expression is

$$\mathcal{T}[\![\mathcal{E}]\!] \Rightarrow \textbf{if}\, \mathcal{T}[\![e_1]\!]\, \textbf{instanceof}\, \sigma(N)\, \textbf{then}\, \mathcal{T}[\![e_2]\!]\, \textbf{else}\, \mathcal{T}[\![e_3]\!]$$

From [I-If] we have that $\Gamma \vdash_c e_1 : \mathsf{nonproxy}\, N; \Gamma_1$, $\Gamma_1 \vdash_i e_2 : T_2; \Gamma_2$ and $\Gamma_1 \vdash_i e_3 : T_3; \Gamma_3$. So, by inductive hypothesis we get $\sigma(\Gamma) \vdash \mathcal{T}[\![e_1]\!] : \mathsf{nonproxy}\, \sigma(N); \sigma(\Gamma_1)$, $\sigma(\Gamma_1) \vdash \mathcal{T}[\![e_2]\!] : \sigma(T_2); \sigma(\Gamma_2)$ and $\sigma(\Gamma_1) \vdash \mathcal{T}[\![e_3]\!] : \sigma(T_3); \sigma(\Gamma_3)$. Furthermore, [I-If] gives $T_2 \leq T$ and $T_3 \leq T$ which means that $\sigma(T_2) \leq \sigma(T)$ and $\sigma(T_3) \leq \sigma(T)$. Also, [I-If] gives $\Gamma' = \mathit{merge}(\Gamma_2, \Gamma_3)$, which means that $\Gamma_2(x) \leq \Gamma'(x)$ and $\Gamma_3(x) \leq \Gamma'(x)$ for all $x \in \mathit{dom}(\Gamma_2) \cap \mathit{dom}(\Gamma_3)$. This implies that $\sigma(\Gamma') = \mathit{merge}(\sigma(\Gamma_2), \sigma(\Gamma_3))$. Therefore, from [If] we get

$$\sigma(\Gamma) \vdash \textbf{if}\, \mathcal{T}[\![e_1]\!]\, \textbf{instanceof}\, \sigma(N)\, \textbf{then}\, \mathcal{T}[\![e_2]\!]\, \textbf{else}\, \mathcal{T}[\![e_3]\!] : \sigma(T); \sigma(\Gamma') \tag{2}$$

As in the previous cases, if $l \in L$, $\mathcal{T}[\![\mathcal{E}^l]\!]$ from equation (2) and [CoerceExpCheck] we get that e typechecks. If $l \notin L$ then $\mathcal{T}[\![\mathcal{E}^l]\!] = \mathcal{T}[\![\mathcal{E}]\!]$ which typecheks from equation (2).

Case [I-Field]: Then $\mathcal{E} \equiv e.f_i$ and the transformed expression is $\mathcal{T}[\![\mathcal{E}]\!] = \mathcal{T}[\![e]\!].f_i$. From [I-Field] we have $\Gamma \vdash_c e : \mathsf{nonproxy}\, N; \Gamma'$, so by inductive hypothesis $\sigma(\Gamma) \vdash \mathcal{T}[\![e]\!] : \mathsf{nonproxy}\, \sigma(N); \sigma(\Gamma')$. [I-Field] also gives $\mathit{fields}(N) = \overline{T}\ \overline{f}$. So, since $\mathcal{T}[\![CT]\!]$ does not change the declared fields, this implies $\mathit{fields}(\sigma(N)) = \sigma(\overline{T})\ f$. Therefore, from [Field] we have

$$\sigma(\Gamma) \vdash \mathcal{T}[\![e]\!].f_i : \sigma(T_i); \sigma(\Gamma') \tag{3}$$

If $l \notin L$ from equation (3) the theorem holds. if $l \in L$ then from [CoerceExpCheck] and equation (3) we have that $\sigma(\Gamma) \vdash \mathcal{T}[\![\mathcal{E}^l]\!] : \sigma(T); \sigma(\Gamma')$.

Case [I-Invoke] then $\mathcal{E} \equiv e_1.m(\overline{e})$, and the transformed expression is $\mathcal{T}[\![\mathcal{E}]\!] \Rightarrow \mathcal{T}[\![e_1]\!].m(\mathcal{T}[\![\overline{e}]\!])$. From [I-Invoke] we have $\Gamma \vdash_c e_1 : \mathsf{nonproxy}\, N; \Gamma'\ \Gamma' \vdash_i \overline{e} : \overline{S}; \Gamma''$ so, by induction hypothesis, $\sigma(\Gamma) \vdash \mathcal{T}[\![e_1]\!] : \mathsf{nonproxy}\, \sigma(N); \sigma(\Gamma')\ \sigma(\Gamma') \vdash \mathcal{T}[\![\overline{e}]\!] : \sigma(\overline{S}); \sigma(\Gamma'')$

Also, from [I-Invoke]: $\mathit{mtype}(m,N) = \overline{T_1} \to Q_1\ \varphi_C^1, \ldots \overline{T_n} \to Q_n\ \varphi_C^n$ and $\overline{S} \leq \overline{T_i}$, $Q_i\ \varphi_C^i \leq \kappa\, \alpha_C$. By definition of *mtype* we have

$$\sigma(\mathit{mtype}(m,N))\mathit{mtype}(m,\sigma(N)) = \sigma(\overline{T_1}) \to \sigma(Q_1)\ \sigma(\varphi_C^1), \ldots \sigma(\overline{T_n}) \to \sigma(Q_n)\ \sigma(\varphi_C^n)$$

and $\sigma(\overline{S}) \leq \sigma(\overline{T_i})$, $\sigma(Q_i)\ \sigma(\varphi^i_C) \leq \sigma(\kappa)\ \sigma(\alpha_C)$ Therefore, from [Invoke] we get

$$\sigma(\Gamma) \vdash \mathcal{T}[\![e_1]\!].m(\mathcal{T}[\![\overline{e}]\!]) : \sigma(\kappa)\ \sigma(\alpha_C); \sigma(\Gamma'') \tag{4}$$

In the case that $l \notin L$ then $\mathcal{T}[\![\mathcal{E}^l]\!] \Rightarrow \mathcal{T}[\![\mathcal{E}]\!]$, so from equation (4) the theorem holds. If $l \in L$ then $\mathcal{T}[\![\mathcal{E}^l]\!] \Rightarrow$ **coerce** $\mathcal{T}[\![\mathcal{E}]\!]$. From equation (4) and [CoerceExpCheck] the theorem holds.

Case [I-New]: $\mathcal{E} \equiv$ **new** $C(\overline{e})$, so $\mathcal{T}[\![\mathcal{E}]\!] \Rightarrow$ **new** $C(\mathcal{T}[\![\overline{e}]\!])$. By hypothesis and [I-New] we know that $\Gamma \vdash_i \overline{e} : \overline{S}; \Gamma'$ so by inductive hypothesis $\sigma(\Gamma) \vdash \mathcal{T}[\![\overline{e}]\!] : \sigma(\overline{S}); \sigma(\Gamma')$

Also, from [I-New] we have $\mathit{fields}(\{C\}_C) = \overline{T}\ \overline{f}$ and $\overline{S} \leq \overline{T}$ so, from the properties of σ: $\sigma(\overline{S}) \leq \sigma(\overline{T})$ and because $\mathcal{T}[\![\cdot]\!]$ does not change field declarations: $\mathit{fields}(\sigma(\{C\}_C)) = \sigma(\overline{T})\ \overline{f}$

Therefore, from [New] we get

$$\sigma(\Gamma) \vdash \textbf{new}\ C(\mathcal{T}[\![\overline{e}]\!]) : \mathsf{nonproxy}\ \sigma(\{C\}_C); \sigma(\Gamma') \tag{5}$$

Because $\mathcal{T}[\![\mathcal{E}]\!]$ typechecks, it follows either directly (if $l \notin L$) or indirectly from [CoerceExpCheck] (if $l \in L$) that $\mathcal{T}[\![\mathcal{E}^l]\!]$ typechecks.

Case [I-Cast]: Then $\mathcal{E} \equiv (N)e$ and $\mathcal{T}[\![\mathcal{E}]\!] \Rightarrow (\sigma(N))\mathcal{T}[\![e]\!]$. From [I-Cast] we know $\Gamma \vdash_c e :$ nonproxy $\varphi'_D; \Gamma'$ so by induction hypothesis $\sigma(\Gamma) \vdash \mathcal{T}[\![e]\!] :$ nonproxy $\sigma(\varphi'_D); \sigma(\Gamma')$ Therefore, one of [UCast], [DCast] or [SCast] always applies.

Case [I-MakeProxy]: Then $\mathcal{E} \equiv$ **makeproxy** e and $\mathcal{T}[\![\mathcal{E}]\!] \Rightarrow$ **makeproxy** $\mathcal{T}[\![e]\!]$. From [I-MakeProxy] we get $\Gamma \vdash_c e :$ nonproxy $N; \Gamma'$ so, by induction hypothesis $\sigma(\Gamma) \vdash \mathcal{T}[\![e]\!] :$ nonproxy $\sigma(N); \sigma(\Gamma')$. Therefore, [MakeProxy] applies, and we have $\sigma(\Gamma) \vdash$ **makeproxy** $\mathcal{T}[\![e]\!] :$ proxy $\sigma(N); \sigma(\Gamma')$.

Case [I-CheckExp]: Then given [I-CheckExp] we have that $\Gamma \vdash_c \mathcal{E}^l :$ nonproxy $N; \Gamma'$ and $\Gamma \vdash_i \mathcal{E}^{l_0} : Q\ N; \Gamma'$. From the induction hypothesis, the latter gives $\sigma(\Gamma) \vdash \mathcal{T}[\![\mathcal{E}]\!] : \sigma(Q)\ \sigma(N); \sigma(\Gamma')$.

In order for the theorem to hold, we must show that $\sigma(\Gamma) \vdash \mathcal{T}[\![\mathcal{E}^l]\!] :$ nonproxy $\sigma(N); \sigma(\Gamma')$ There are two cases.

- $l \notin L$. Then [I-CheckExp] gives that $l \notin L \Rightarrow$ proxy $\not\leq Q$. Moreover, $\sigma(Q)$ is either proxy or nonproxy, $\sigma($proxy$) =$ proxy and $\sigma($nonproxy$) =$ nonproxy. Finally, since proxy $\neq$ nonproxy, we derive that $\sigma(Q) =$ nonproxy. This is proved by contradiction; if $\sigma(Q) =$ proxy, then we would have proxy $\not\leq$ proxy.
  In this case $\mathcal{T}[\![\mathcal{E}^l]\!] \Rightarrow \mathcal{T}[\![\mathcal{E}]\!]$, so the theorem holds, since $\sigma(Q) =$ nonproxy.
- $l \in L$. In this case, $\mathcal{T}[\![\mathcal{E}^l]\!] \Rightarrow$ **coerce** $\mathcal{T}[\![\mathcal{E}]\!]$. Therefore, from [CoreceExpCheck] and $\sigma(\Gamma) \vdash \mathcal{T}[\![\mathcal{E}]\!] : \sigma(Q)\ \sigma(N); \sigma(\Gamma')$ we get that the theorem holds.

Case [I-CheckVar]: In this case, $\mathcal{E} \equiv x$, so $\mathcal{T}[\![\mathcal{E}]\!] \Rightarrow x$. [I-CheckVar] gives $\Gamma[x \mapsto Q\ N] \vdash_i x^{l_0} : Q\ N; \Gamma[x \mapsto Q\ N]$ which, by induction hypothesis, implies

$$\sigma(\Gamma)[x \mapsto \sigma(Q)\ \sigma(N)] \vdash \mathcal{T}[\![x]\!] : \sigma(Q)\ \sigma(N); \sigma(\Gamma)[x \mapsto \sigma(Q)\ \sigma(N)] \tag{6}$$

Similarly with [I-CheckExp], there are two cases:

- $l \notin L$ Then as before, $\sigma(Q)$ must be nonproxy, otherwise we would have a contradiction proxy $\not\leq$ proxy from proxy $\leq Q \Rightarrow l \in L$. In this case equation (6) becomes $\sigma(\Gamma)[x \mapsto$ nonproxy $\sigma(N)] \vdash \mathcal{T}[\![x]\!] :$ nonproxy $\sigma(N); \sigma(\Gamma)[x \mapsto$ nonproxy $\sigma(N)]$, so the theorem holds.

- $l \in L$ Then $\mathcal{T}[\![\mathcal{E}^l]\!] \Rightarrow$ **coerce** x. Therefore, from [CoerceVarCheck] and equation (6) we have that the theorem holds.

Case [I-Method] Given that [I-Method] holds and typechecks method $\mathcal{M}$, we will show that $\mathcal{T}[\![\mathcal{M}]\!]$ typechecks given the solution σ, L.

The transformation gives $\mathcal{T}[\![S\ m(\overline{T}\ \overline{x})\ \{\ \textbf{return}\ e;\}]\!] \Rightarrow \sigma(S)\ m(\sigma(\overline{T})\ \overline{x})\ \{\ \textbf{return}\ \mathcal{T}[\![e]\!];\}$. [I-Method] gives: $\overline{x}:\overline{T}, \textbf{this} : \mathsf{nonproxy}\, C \vdash_i e : U; \Gamma' \; U \leq S \; CT(C) = \textbf{class}\, C\ \textbf{extends}\ D\ \{\ldots;\ldots\}$ $override(m, D, \overline{T} \rightarrow S)$ By induction hypothesis $\overline{x} : \sigma(\overline{T}), \textbf{this} : \mathsf{nonproxy}\ \sigma(C) \vdash \mathcal{T}[\![e]\!] : \sigma(U); \sigma(\Gamma')$ Also $\sigma(U) \leq \sigma(S)\ \mathcal{T}[\![CT(C)]\!] = \textbf{class}\, C\ \textbf{extends}\ D\ \{\ldots;\ldots\}$ $override(m, D, \sigma(\overline{T}) \rightarrow \sigma(S))$ Therefore by [Method] $\vdash \sigma(S)\ m(\sigma(\overline{T})\ \overline{x})\ \{\ \textbf{return}\ \mathcal{T}[\![e]\!];\}$

Case [I-Class] Given that [I-Class] holds and typechecks class $\mathcal{C}$, we will show that $\mathcal{T}[\![\mathcal{C}]\!]$ typechecks given the solution σ, L.

The transformation gives $\mathcal{T}[\![\textbf{class}\, C\ \textbf{extends}\ D\ \{\ \overline{T}\ \overline{f}; K\ \overline{M}\ \}]\!] \Rightarrow \mathcal{T}[\![\textbf{class}\, C\ \textbf{extends}\ D\ \{\ \sigma(\overline{T})\ \overline{f}; \mathcal{T}[\![K]\!]\ \mathcal{T}[\![\overline{M}]\!]\ \}]\!]$ [I-Class] gives: $K = C(\overline{T}\ \overline{g}, \overline{S}\ \overline{f})\ \{\ \textbf{super}(\overline{g});\ \textbf{this}.\overline{f} = \overline{f};\ \}$ $fields(D) = \overline{T}\ \overline{g}\ \overline{T} = \kappa_1\ \alpha^1_{C_1}, \ldots \kappa_n\ \alpha^n_{C_n}$ $\overline{S} = \kappa'_1\ \alpha'^1_{C_1}, \ldots \kappa'_n\ \alpha'^n_{C_n}\ \kappa_i, \kappa'_i, \alpha^i, \alpha'^i$ fresh $\vdash_i \overline{M}$

By induction hypothesis, we have that $\vdash \mathcal{T}[\![\overline{M}]\!]$. Also, $\sigma(fields(D)) = \sigma(\overline{T})\ \overline{g}$. Moreover $\mathcal{T}[\![K]\!]$ gives $\mathcal{T}[\![C(\overline{T}\ \overline{g}, \overline{S}\ \overline{f})\ \{\ \textbf{super}(\overline{g});\ \textbf{this}.\overline{f} = \overline{f};\ \}]\!] \Rightarrow C(\sigma(\overline{T})\ \overline{g}, \sigma(\overline{S})\ \overline{f})\ \{\ \textbf{super}(\overline{g});\ \textbf{this}.\overline{f} = \overline{f};\ \}$ So, [Class] implies $\vdash \textbf{class}\, C\ \textbf{extends}\ D\ \{\ \sigma(\overline{T})\ \overline{f}; \mathcal{T}[\![K]\!]\ \mathcal{T}[\![\overline{M}]\!]\ \}$

□