Contents

1 Type Systems: Advances and Applications **1**

1.1 Introduction 1

1.2 Types for Confinement 3

1.2.1 Background 4

1.2.2 Static Analysis 5

1.2.3 Confined Types 6

1.2.4 Related Work on Alias Control 20

1.3 Type Qualifiers 21

1.3.1 Background 21

1.3.2 Static Analysis 22

1.3.3 A Type System for Qualifiers 23

1.3.4 Related Work on Type Refinements 34

Chapter 1

Type Systems: Advances and Applications

1.1 Introduction

This chapter is about the convergence of type systems and static analysis. Historically, these two approaches to reasoning about programs have had different purposes. Type systems were developed in order to catch common kinds of programming errors early in the software development cycle. In contrast, static analyses were developed in order to automatically optimize the code generated by a compiler. The two fields also have different theoretical foundations: type systems are typically formalized as logical inference systems [65], while static analyses are typically formalized as abstract program executions [49, 23].

Recently, however, there has been a convergence of the objectives and techniques underlying type systems and static analysis [61, 45, 59, 62]. On the one hand, static analysis is increasingly being used for program understanding and error detection, rather than purely for code optimization. For example, the LCLint tool [33] uses static analysis to detect null-pointer dereferences and other common errors in C programs, and it relies on type-system-like program annotations for efficiency and precision. As another example, the ESP

tool [24] uses static analysis to detect violations of API usage protocols, for example that a file can only be read or written after it has been opened.

On the other hand, type systems have become a mature and widely accepted technology. Programmers write most new software in languages such as C [48], C++ [32], Java [42], and C# [53], which all feature varying degrees of static type checking. For example, the Java type system guarantees that if a program calls a method on some object, then at run time the object will actually have a method of that name, expecting the proper number and kind of arguments. Types are also used in the intermediate languages of compilers and even in assembly languages [55], such as the typed assembly language for x86 called TALx86 [54].

With this success, researchers have been motivated to explore the potential to extend traditional type systems to detect a variety of interesting classes of program errors. This exploration has shown type systems to be a robust approach to static reasoning about programs and their properties. For example, type systems have been recently used to ensure the safety of manual memory management (e.g., [68, 57, 43]), to track and restrict the aliasing relationships among pointers (e.g., [22, 13, 1, 34]), and to ensure the proper interaction of threads in concurrent programs (e.g., [35, 10, 36]).

These new uses of type systems have brought type systems closer to the domain of static analysis, both in terms of objectives and techniques. For example, reasoning about aliasing is traditionally done via a static analysis to compute the set of *may-aliases*, rather than via a type system. As another example, some sophisticated uses of type systems have required making types *flow-sensitive* [26, 40], whereby the type of an expression can change at each program point (e.g., a file's type might denote that the file is open at one point but closed at another point). This style of type system has a natural relationship to traditional static analysis, where the set of "flow facts" can change at each program point.

In this chapter, we describe two type systems that both have a strong relationship to static analysis. Each of the type systems is a *refinement* of an existing and well-understood type system: the first refines a subset of the Java type system while the second refines a system of simple types for the lambda calculus. The refinements are done via annotations that refine existing types in order to specify and check finer-grained properties. Many of

the sophisticated type systems mentioned above can be viewed as refinements of existing types and type systems. Such type systems are examples of *type-based analyses* [60]; that is, they assume and leverage the existing type system and they provide information only for programs that type check with the existing type system.

In the following section we describe a type system that ensures a strong form of encapsulation in object-oriented languages. Namely, the analysis guarantees that an object of a class declared `confined` will never dynamically escape the class's scope. Object confinement goes well beyond the guarantees of traditional privacy modifiers like `protected` and `private`, and it bears a strong relationship to standard static analyses.

Language designers cannot anticipate all of the refinements that will be useful for programmers nor all of the ways in which these refinements can be used to practically check programs. Therefore, it is desirable to provide a framework that allows programmers to easily augment a language's type system with new refinements of interest for their applications. In the third section we describe a representative framework of this kind, supporting programmer-defined *type qualifiers*. A type qualifier is a simple but useful kind of type refinement consisting solely of an uninterpreted "tag." For example, C's `const` qualifier refines an existing type to further indicate that values of this type are not modifiable, and a `nonnull` qualifier could refine a pointer type to further indicate that pointers of this type are never null.

1.2 Types for Confinement

In this section we will use types to ensure that an object cannot escape the scope of its class. Our presentation is based on results from three papers on *confined types* [9, 44, 75].

1.2.1 Background

Object-oriented languages such as Java provide a way of protecting the name of a field, but not the contents of a field. Consider the following example.

```
package p;

public class Table {
   private Bucket[] buckets;
   public Object[] get(Object key) { return buckets; }
}

class Bucket {
   Bucket next;
   Object key, val;
}
```

The hash table class `Table` is a public class which uses a package-scoped class `Bucket` as part of its implementation. The programmer has declared the field `buckets` to be private and intends the hash-table-bucket objects to be internal data structures which should not escape the scope of the `Bucket` class. The declaration of `Bucket` as packaged scoped ensures that the `Bucket` class is not visible outside the package `p`. However, even the combination of a private field and a package-scoped class does *not* prevent `Bucket` objects from being accessible outside the scope of the `Bucket` class. To see why, notice that the public `get` method in class `Table` has body `return buckets;` which provides an array of bucket objects to any client, including clients outside the package `p`. Any client can now update the array and thereby change the behavior of the hash table.

The example shows how an object reference can leak out of a package. Such leakage is a problem because (1) the object may represent private information such as a private key and (2) code outside the package may update the object, making it more difficult for programmers

to reason about the program. The problem stems from a combination of aliasing and side effects. Aliasing occurs when an object is accessible through different access paths. In the above example, code outside the package can access bucket objects and update them.

How can we ensure that an object cannot escape the scope of its class? We will briefly discuss how one can solve the problem using static analysis and then proceed to show a type-based solution.

1.2.2 Static Analysis

Static analysis can be used to determine whether an object can escape the scope of its class. We will explain a whole-program analysis, that is, an approach which requires access to all the code in the application and its libraries.

Assuming that we have the whole program, let `U` be the set of class names in the program. The basic idea is to statically compute, for each expression e in the program, a subset of `U` which conservatively approximates the possible values of e. We will call that set the *flow set* for e. For example, if the flow set for e is the set $\{A, B, C\}$, then that means that the expression e will evaluate to either an A-object, a B-object, or a C-object. Notice that we allow the set to be a conservative approximation; for example, e might never evaluate to a C-object. All we require is that if e evaluates to an X-object, then X is a member of the flow set for e.

Researchers have published many approaches to statically computing flow sets for expressions in object-oriented programs, see for example [63, 25, 3, 67, 70] for some prominent and efficient whole-program analyses. For the purposes of our discussion here, all we rely on is that flow sets can be computed statically.

Once we have computed flow sets, we can for each package-scoped class C determine whether C ever appears in the flow set for an expression *outside* the package of C. For each class that never appears in flow sets outside its package, we know that its objects don't escape its package in this particular program.

The whole-program-analysis approach has several drawbacks.

Bug finding after the program is done. First, the approach finds bugs *after* the whole program is done. While that is useful, we would like to help the programmer find bugs *while* he/she writes the program.

No enforcement of discipline. Second, the static analysis does not enforce any discipline on the programmer. A programmer can write crazy code and the static analysis may then simply report that every object can escape the scope of its class. While that should be a red flag for the programmer, we would like to help the programmer determine which lines of code to fix to avoid some of the problems.

Fragile. Third, the static analysis tends to be sensitive to small changes in the program text. For one version of a program, a static analysis may find no problems with escaping objects, and then after a few lines of changes, suddenly the static analysis finds problems all over the place. We would like to help the programmer build software in a modular way such that changes in one part of the program do not affect other parts of the program.

The type-based approach in the next section has none of the three drawbacks.

The static-analysis approach in this section is one among many static analyses that solve the same or similar problems. For example, researchers have published powerful escape analyses [6, 7, 8, 30] some of which can be adapted to the problem we consider in this chapter.

1.2.3 Confined Types

We can use types to ensure that an object cannot escape the scope of its class. We will show an approach for Java which extends Java with the notions of *confined type* and *anonymous method*. The idea is that if we declare a class to be confined, then the type system will enforce rules that ensure that an object of the class cannot escape the scope of the class. If a program type checks in the extended type system, then an object cannot escape the scope of its class.

Confinement can be enforced using two sets of constraints. The first set of constraints, *confinement rules*, applies to the classes defined in the same package as the confined class. These rules track values of confined types and ensure that they are neither exposed in public members, nor widened to non-confined types.

The second kind of constraints, *anonymity rules*, applies to methods inherited by the confined classes, potentially including library code, and ensures that these methods do not leak a reference to the distinguished variable `this` which may refer to an object of confined type.

We will discuss the confinement and anonymity rules next, and later show how to formalize the rules and integrate them into the Java type system.

Confinement Rules

The following confinement rules must hold for all classes of a package containing confined types.

- $\mathcal{C}1$: A confined type must not appear in the type of a public (or protected) field or the return type of a public (or protected) method.
- $\mathcal{C}2$: A confined type must not be public.
- $\mathcal{C}3$: Methods invoked on an expression of confined type must either be defined in a confined class or be anonymous methods.
- $\mathcal{C}4$: Subtypes of a confined type must be confined.
- $\mathcal{C}5$: Confined types can be widened only to other confined types.
- $\mathcal{C}6$: Overriding must preserve anonymity of methods.

Rule $\mathcal{C}1$ prevents exposure of confined types in the public interface of the package as client code could break confinement by accessing values of confined types through a type's public interface. Rule $\mathcal{C}2$ is needed to ensure that client code cannot instantiate a confined class.

It also prevents client code from declaring field or variables of confined types. The latter restriction is needed so that code in a confining package will not mistakenly assign objects of confined types to the fields or variables outside that package. Rule $\mathcal{C}3$ ensures that methods invoked on an object enforce confinement. In the case of methods defined in the confining package, this ensues from the other confinement rules. Inherited methods defined in another package do not have access to any confined fields, since those are package-scoped (Rule $\mathcal{C}1$). However, an inherited method of confined class may leak the `this` reference, which is implicitly widened to the method's declaring class. To prevent this, Rule $\mathcal{C}3$ requires these methods to be anonymous (as explained below). Rule $\mathcal{C}4$ prevents the declaration of a public subclass of a confined type. This prevents *spoofing* leaks where a public subtype defined outside of the confined package is used to access private fields [19]. Rule $\mathcal{C}5$ prevents code within confining packages from assigning values of confined types to fields or variables of public types. Finally, Rule $\mathcal{C}6$ allows us to statically verify the anonymity of the methods that are invoked on expressions of confined types.

Anonymity Rule

The anonymity rule applies to inherited methods which may reside in classes outside of the enclosing package. This rule prevents a method from leaking the `this` reference. A method is *anonymous* if it has the following property.

- $\mathcal{A}1$: The `this` reference is used only to select fields and as the receiver in the invocation of other anonymous methods.

This prevents an inherited method from storing or returning `this` as well as using it as an argument to a call. Selecting a field is always safe, as it cannot break confinement because only the fields visible in the current class can be accessed. Method invocation (on `this`) is restricted to other methods that are anonymous as well. Note that we check this constraint assuming the static type of `this` and Rule $\mathcal{C}6$ ensures that the actual method invoked on `this` will also be anonymous. Thus, Rule $\mathcal{C}6$ ensures that the anonymity of a method is independent of the result of method lookup.

Rule $\mathcal{C}6$ could be weakened to apply only to methods inherited by confined classes. For instance, if an anonymous method `m` of class `A` is overridden in both class `B` and `C`, and `B` is extended by a confined class while `C` is not, then the method `m` in `B` must be anonymous while `m` of `C` needs not be. The reason is that the method `m` of `C` will never be invoked on confined objects and thus there is no need for it to be anonymous.

Confined Featherweight Java

Confined Featherweight Java, which we refer to as ConfinedFJ, is a minimal core calculus for modeling confinement for a Java-like object-oriented language. ConfinedFJ extends Featherweight Java (FJ) which was designed by Igarashi, Pierce and Wadler [46] to model the Java type system. It is a core calculus as it limits itself to a subset of the Java language with the following five basic expressions: object construction, method invocation, field access, casts and local variable access. This spartan setting has proved appealing to researchers. ConfinedFJ stays true to the spirit of FJ. The surface differences lie in the presence of class and method level visibility annotations. In ConfinedFJ, classes can be declared to be either public or confined, and methods can optionally be declared as anonymous. One further difference is that ConfinedFJ class names are pairs of identifiers bundling a package name and a class name just as in Java.

Syntax

Let metavariable `L` range over class declarations, $\mathtt{C}, \mathtt{D}, \mathtt{E}$ range over a denumerable set of class identifiers, $\mathtt{K}, \mathtt{M}$ range over constructor and method declarations respectively, and `f` and `x` range over field names and variables (including parameters and the pseudo-variable `this`) respectively. Let $\mathtt{e}, \mathtt{d}$ range over expressions and $\mathtt{u}, \mathtt{v}, \mathtt{w}$ range over values.

We adopt FJ notational idiosyncrasies and use an over-bar to represent a finite (possibly empty) sequence. We write $\overline{\mathtt{f}}$ to denote the sequence $\mathtt{f}_1, \ldots, \mathtt{f}_n$ and similarly for $\overline{\mathtt{e}}$ and $\overline{\mathtt{v}}$. We write $\overline{\mathtt{C}\,\mathtt{f}}$ to denote $\mathtt{C}_1\,\mathtt{f}_1, \ldots \mathtt{C}_n\,\mathtt{f}_n$, $\overline{\mathtt{C}} <: \overline{\mathtt{D}}$ to denote $\mathtt{C}_1 <: \mathtt{D}_1, \ldots, \mathtt{C}_n <: \mathtt{D}_n$ and finally $\mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}}$ to denote $\mathtt{this}.\mathtt{f}_1 = \mathtt{f}_1, \ldots, \mathtt{this}.\mathtt{f}_n = \mathtt{f}_n$.

$$
\begin{array}{lcl}
\texttt{C} & ::= & \texttt{p.q} \\
\texttt{L} & ::= & [\texttt{public}|\texttt{conf}]\ \texttt{class C} \lhd \texttt{D}\ \{\ \overline{\texttt{C f}};\ \texttt{K}\ \overline{\texttt{M}}\ \} \\
\texttt{K} & ::= & \texttt{C}(\overline{\texttt{C f}})\ \{\ \texttt{super}(\overline{\texttt{f}});\ \texttt{this.}\overline{\texttt{f}} = \overline{\texttt{f}};\ \} \\
\texttt{M} & ::= & [\texttt{anon}]\ \texttt{C m}(\overline{\texttt{C x}})\ \{\ \texttt{return e};\ \} \\
\texttt{e} & ::= & \texttt{x} \mid \texttt{e.f} \mid \texttt{e.m}(\overline{\texttt{e}}) \mid (\texttt{C})\ \texttt{e} \mid \texttt{new C}(\overline{\texttt{e}}) \\
\texttt{v} & ::= & \texttt{new C}(\overline{\texttt{v}})
\end{array}
$$

Figure 1.1: ConfinedFJ: Syntax.

The syntax of ConfinedFJ is given in Figure 1.1. An expression `e` can be either one of a variable `x` (including `this`), a field access `e.f`, a method invocation $\texttt{e.m}(\overline{\texttt{e}})$, a cast `(C) e`, an object $\texttt{new C}(\overline{\texttt{e}})$. Since ConfinedFJ has a call-by-value semantics, it is expedient to add a special syntactic form for fully evaluated objects, denoted $\texttt{new C}(\overline{\texttt{v}})$.

Class identifiers are pairs `p.q` such that `p` and `q` range over denumerable disjoint sets of names. For ConfinedFJ class name `p.q`, `p` is interpreted as a *package name* and `q` as a *class name*. In ConfinedFJ, class identifiers are fully qualified. For a class identifier `C`, *packof*(`C`) denotes the identifier's package prefix, so, for example, the value of *packof*(`p.O`) is `p`.

Each class declarations is annotated with one of the visibility modifiers `public`, `conf`, or none; a public class is declared by `public class C` $\lhd$ `D` $\{\ldots\}$, a package-scoped, confined class is `conf class C` $\lhd$ `D`$\{\ldots\}$, and a package-scoped, nonconfined class is `class C` $\lhd$ `D`$\{\ldots\}$. Methods can be annotated with the optional `anon` modifier to denote anonymity.

We will not formalize the dynamic semantics of ConfinedFJ (for full details, see [75]). We assume a class table CT which stores the definitions of all classes of ConfinedFJ program such that CT(`C`) is the definition of class `C`. The subtyping relation `C` $<:$ `D` denotes that class `C` is a subtype of class `D`; $<:$ is the smallest reflexive and transitive class ordering that has the property that if `C` extends `D`, then `C` $<:$ `D`. Every class is a subtype of `l.Object`. The function *fields*(`C`) return the list of all fields of the class `C` including inherited ones; *methods*(`C`) returns the list of all methods in the class `C`; *mdef*(`m`) returns the identifier of defining class for the method `m`.

Type Rules

Figure 1.2 defines relations used in the static semantics. The predicate *conf*(C) holds if the class table maps C to a class declared as confined. Similarly, the predicate *public*(C) holds if the class table maps C to a class declared as public. The function *mtype*(m, C) yields the type signature of a method. The predicate *override*(m, C, D) holds if m is a valid, anonymity preserving, redefinition of an inherited method or if this is the method's original definition. Class visibility, written *visible*(C, D), states that a class C is visible from D if, either, C is public, or if both classes are in the same package.

The *safe subtyping* relation, written $\texttt{C} \preceq \texttt{D}$, is a confinement preserving restriction of the subtyping relation <:. A class C is a safe subtype of D if C is a subtype of D, and either C is public or D is confined. This relation is used in the typing rules to prevent widening a confined type to a public type; confinement-preserving widening requires safe subtyping to hold. The type system further constrains subtyping by enforcing that all subclasses of a confined class must belong to the same package (see the T-Class rule and the definition of visibility). Notice that safe subtyping is reflexive and transitive.

Figure 1.3 defines constraints imposed on anonymous methods. A method m is anonymous in class C, written *anon*(m, C), if its declaration is annotated with the `anon` modifier. The following syntactic restrictions are imposed on the body of an anonymous method. An expression e is anonymous in class C, written *anon*(e, C), if the pseudo-variable `this` is used solely for field selection and anonymous method invocation. (C) e is anonymous if e is anonymous. `new` $\texttt{C}(\overline{\texttt{e}})$ and $\texttt{e.m}(\overline{\texttt{e}})$ are anonymous if $\texttt{e} \neq \texttt{this}$ and $\texttt{e}, \overline{\texttt{e}}$ are anonymous. With the exception of `this` all variables are anonymous. `this.f` is always anonymous, and $\texttt{this.m}(\overline{\texttt{e}})$ is anonymous in C if m is anonymous in C and $\overline{\texttt{e}}$ is anonymous. We write *anon*($\overline{\texttt{e}}$, C) to denote that all expressions in $\overline{\texttt{e}}$ are anonymous.

Expression typing rules

The typing rules for ConfinedFJ are given in Figure 1.4, where type judgments have the form $\Gamma \vdash \texttt{e} : \texttt{C}$, in which Γ is an environment that maps variables to their types. The main

Confined types, type visibility, and safe subtyping:

$$\frac{CT(\texttt{C}) = \texttt{conf class C} \triangleleft \texttt{D}\ \{\ldots\}}{conf(\texttt{C})}$$

$$\frac{public(\texttt{C})}{visible(\texttt{C},\texttt{D})} \qquad \frac{packof(\texttt{C}) = packof(\texttt{D})}{visible(\texttt{C},\texttt{D})}$$

$$\frac{\texttt{C} <: \texttt{D} \quad conf(\texttt{C}) \Rightarrow conf(\texttt{D})}{\texttt{C} \preceq \texttt{D}}$$

Method type lookup:

$$\frac{mdef(\texttt{m},\ \texttt{C}) = \texttt{D} \quad [\texttt{anon}]\ \texttt{B}\ \texttt{m}(\overline{\texttt{B x}})\ \{\,\texttt{return e};\,\} \in methods(\texttt{D})}{mtype(\texttt{m},\ \texttt{C}) = \overline{\texttt{B}} \rightarrow \texttt{B}}$$

Valid method overriding:

$$\frac{\begin{array}{c}\text{either m is not defined in D or any of its parents, or} \\ mtype(\texttt{m},\ \texttt{C}) = \overline{\texttt{C}} \rightarrow \texttt{C}_0 \quad mtype(\texttt{m},\ \texttt{D}) = \overline{\texttt{C}} \rightarrow \texttt{C}_0 \quad (anon(\texttt{m},\texttt{D}) \Rightarrow anon(\texttt{m},\texttt{C}))\end{array}}{override(\texttt{m},\texttt{C},\texttt{D})}$$

Figure 1.2: ConfinedFJ: Auxiliary definitions.

Anonymous method:

$$\frac{mdef(\texttt{m},\ \texttt{C}_0) = \texttt{C}_0' \quad \texttt{anon C m}\ (\overline{\texttt{C x}})\ \{\ldots\} \in methods(\texttt{C}_0')}{anon(\texttt{m},\texttt{C}_0)}$$

Anonymity constraints:

$$\frac{anon(\texttt{e},\texttt{C})}{anon((\texttt{C}')\ \texttt{e},\texttt{C})} \qquad \frac{anon(\overline{\texttt{e}},\texttt{C})}{anon(\texttt{new C}'(\overline{\texttt{e}}),\texttt{C})} \qquad \frac{\texttt{x} \neq \texttt{this}}{anon(\texttt{x},\texttt{C})}$$

$$\frac{anon(\texttt{e},\texttt{C})}{anon(\texttt{e.f},\texttt{C})} \qquad \frac{anon(\texttt{e},\texttt{C}) \quad anon(\overline{\texttt{e}},\texttt{C})}{anon(\texttt{e.m}(\overline{\texttt{e}}),\texttt{C})}$$

$$\frac{}{anon(\texttt{this.f},\texttt{C})} \qquad \frac{anon(\texttt{m},\texttt{C}) \quad anon(\overline{\texttt{e}},\texttt{C})}{anon(\texttt{this.m}(\overline{\texttt{e}}),\texttt{C})}$$

Figure 1.3: ConfinedFJ: Syntactic Anonymity Constraints.

Expression typing:

$$\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \qquad \text{(T-Var)}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{C} \quad \mathit{fields}(\mathtt{C}) = (\overline{\mathtt{C}\ \mathtt{f}})}{\Gamma \vdash \mathtt{e.f_i} : \mathtt{C_i}} \qquad \text{(T-Field)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e} : \mathtt{C}_0 \quad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}} \quad \mathit{mtype}(\mathtt{m},\ \mathtt{C}_0) = \overline{\mathtt{D}} \to \mathtt{C} \quad \overline{\mathtt{C}} \preceq \overline{\mathtt{D}} \\ \mathit{mdef}(\mathtt{m},\ \mathtt{C}_0) = \mathtt{D}_0 \quad (\mathtt{C}_0 \preceq \mathtt{D}_0 \vee \mathit{anon}(\mathtt{m}, \mathtt{D}_0))\end{array}}{\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{C}} \qquad \text{(T-Invk)}$$

$$\frac{\mathit{fields}(\mathtt{C}) = (\overline{\mathtt{D}\ \mathtt{f}}) \quad \Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{C}} \quad \overline{\mathtt{C}} \preceq \overline{\mathtt{D}}}{\Gamma \vdash \mathtt{new\ C}(\overline{\mathtt{e}}) : \mathtt{C}} \qquad \text{(T-New)}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{D} \quad \mathit{conf}(\mathtt{D}) \Rightarrow \mathit{conf}(\mathtt{C})}{\Gamma \vdash (\mathtt{C})\ \mathtt{e} : \mathtt{C}} \qquad \text{(T-UCast)}$$

Method typing:

$$\frac{\begin{array}{c}\overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C}_0 \vdash \mathtt{e} : \mathtt{D} \quad \mathtt{D} \preceq \mathtt{C} \quad \mathit{override}(\mathtt{m}, \mathtt{C}_0, \mathtt{D}_0) \\ \overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C}_0 \vdash \mathit{visible}(\mathtt{e}, \mathtt{C}_0) \quad (\mathit{anon}(\mathtt{m}, \mathtt{C}_0) \Rightarrow \mathit{anon}(\mathtt{e}, \mathtt{C}_0))\end{array}}{[\mathtt{anon}]\ \mathtt{C}\ \mathtt{m}(\overline{\mathtt{C}\ \mathtt{x}})\ \{\ \mathtt{return\ e};\ \}\ \mathtt{OK\ IN}\ \mathtt{C}_0 \triangleleft \mathtt{D}_0} \qquad \text{(T-Method)}$$

Class typing:

$$\frac{\begin{array}{c}\mathit{fields}(\mathtt{D}) = (\overline{\mathtt{D}\ \mathtt{g}}) \quad \mathtt{K} = \mathtt{C}(\overline{\mathtt{D}\ \mathtt{g}},\ \overline{\mathtt{C}\ \mathtt{f}})\ \{\mathtt{super}(\overline{\mathtt{g}});\ \mathtt{this.}\overline{\mathtt{f}} = \overline{\mathtt{f}};\} \\ \mathit{visible}(\mathtt{D}, \mathtt{C}) \quad (\mathit{conf}(\mathtt{D}) \Rightarrow \mathit{conf}(\mathtt{C})) \quad \overline{\mathtt{M}}\ \mathtt{OK\ IN}\ \mathtt{C} \triangleleft \mathtt{D}\end{array}}{[\mathtt{public}|\mathtt{conf}]\ \mathtt{class\ C} \triangleleft \mathtt{D}\ \{\ \overline{\mathtt{C}\ \mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\ \}\ \mathtt{OK}} \qquad \text{(T-Class)}$$

Static expression visibility:

$$\frac{\mathit{visible}(\Gamma(\mathtt{x}), \mathtt{C})}{\Gamma \vdash \mathit{visible}(\mathtt{x}, \mathtt{C})} \qquad \frac{\Gamma \vdash \mathtt{e.f_i} : \mathtt{C}' \quad \mathit{visible}(\mathtt{C}', \mathtt{C}) \quad \Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{C})}{\Gamma \vdash \mathit{visible}(\mathtt{e.f_i}, \mathtt{C})}$$

$$\frac{\mathit{visible}(\mathtt{C}', \mathtt{C}) \quad \Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{C})}{\Gamma \vdash \mathit{visible}((\mathtt{C}')\ \mathtt{e}, \mathtt{C})} \qquad \frac{\mathit{visible}(\mathtt{C}', \mathtt{C}) \quad \forall \mathtt{i},\ \Gamma \vdash \mathit{visible}(\mathtt{e_i}, \mathtt{C})}{\Gamma \vdash \mathit{visible}(\mathtt{new\ C}'(\overline{\mathtt{e}}), \mathtt{C})}$$

$$\frac{\Gamma \vdash \mathtt{e.m}(\overline{\mathtt{e}}) : \mathtt{C}' \quad \mathit{visible}(\mathtt{C}', \mathtt{C}) \quad \Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{C}) \quad \forall \mathtt{i},\ \Gamma \vdash \mathit{visible}(\mathtt{e_i}, \mathtt{C})}{\Gamma \vdash \mathit{visible}(\mathtt{e.m}(\overline{\mathtt{e}}), \mathtt{C})}$$

Figure 1.4: ConfinedFJ: Typing rules.

difference with FJ is that these rules disallow unsafe widening of types. This is captured by conditions of the form $\mathtt{C} \preceq \mathtt{D}$ which enforce safe subtyping.

- Rules T-VAR and T-FIELD are standard.
- Rule T-NEW prevents instantiating an object if any of the object's fields with a public type is given a confined argument. That is, for fields with declared types $\overline{\mathtt{D}}$ and argument types $\overline{\mathtt{C}}$, relation $\overline{\mathtt{C}} \preceq \overline{\mathtt{D}}$ must hold. By definition of $\mathtt{C_i} \preceq \mathtt{D_i}$, if $\mathtt{C_i}$ is confined then $\mathtt{D_i}$ is confined as well.
- Rule T-INVK prevents widening of confined arguments to public parameters by enforcing safe subtyping of argument types with respect to parameter types. In order to prevent implicit widening of the receiver, we consider two cases. Assume that the receiver has type $\mathtt{C_0}$ and the method `m` is defined in $\mathtt{D_0}$, then it must either be the case that $\mathtt{C_0}$ is a safe subtype of $\mathtt{D_0}$ or that `m` has been declared anonymous in $\mathtt{D_0}$.
- Rule T-UCAST prevents casting a confined type to a public type. Notice that a down cast preserves confinement because by Rule T-CLASS a confined class can only have confined subclasses.

Typing rules for methods and classes

Figure 1.4 also gives rules for typing methods and classes.

- Rule T-METHOD places the following constraints on a method `m` defined in class $\mathtt{C_0}$ with body `e`. The type `D` of `e` must be a safe subtype of the method's declared type `C`. The method must preserves anonymity declarations. If `m` is declared anonymous, `e` must comply with the corresponding restrictions. The most interesting constraint is the visibility enforced on the body by $\Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{C_0})$, which is defined recursively over the structure of terms. It ensures that the types of all subexpressions of `e` are visible from the defining class $\mathtt{C_0}$. In particular, the method parameters used in the method body `e` must have types visible in $\mathtt{C_0}$.

- Rule T-Class requires that if class `C` extends `D` then `D` be visible in `C` and if `D` is confined, then so is `C`. Rule T-Class allows the fields of a class `C` to have types not visible in `C`, but the constraint of $\Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{C})$ in Rule T-Method prohibits the method of `C` from accessing such fields.

The class table CT is well-typed if all classes in CT are well-typed. For the rest of this paper, we assume CT to be well-typed.

Relation to the Informal Rules

We now relate the confinement and anonymity rules with the ConfinedFJ type system. The effect of Rule $\mathcal{C}1$, which limits the visibility of fields if their type is confined, is obtained as a side effect of the visibility constraint as it prevents code defined in another package from accessing a confined field. ConfinedFJ could be extended with field and method access modifier without significantly changing the type system. The expression typing rules enforce confinement Rules $\mathcal{C}3$ and $\mathcal{C}5$ by ensuring that methods invoked on an object of confined type are either anonymous or defined in a confined class, and that widening is confinement preserving. Rule $\mathcal{C}2$ uses access modifiers to limit the use of confined types; and the same effect is achieved by the visibility constraint $\Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{C})$ on expression part of T-Method. Rule $\mathcal{C}4$, which states that subclassing is confinement preserving, is enforced by T-Class. Rule $\mathcal{C}6$, which states that overriding is anonymity preserving, is enforced by T-Method. Finally the anonymity constraint of Rule $\mathcal{A}1$ is obtained by the *anon* predicate in the antecedent of T-Method.

Two ConfinedFJ Examples

Consider the following stripped down version of a hash table class written in ConfinedFJ. The hash table is represented by a class `p.Table` defined in some package `p` that holds a single bucket of class `p.Buck`. The bucket can be obtained by calling the method `get()` on a table, the bucket's data can then be obtained by calling `getData()`. In this example, buckets are confined but they extend a public class `p.Cell`. The interface of `p.Table.get()`

```
class p.Table ◁ l.Object {
   p.Buck buck;
   Table(p.Buck buck) { super(); this.buck = buck; }
   p.Cell get() { return this.buck; }
}
class p.Cell ◁ l.Object {
   l.Object data;
   l.Object getData() { return this.data; }
}
conf class p.Buck ◁ p.Cell {
   p.Buck() { super(); }
}
class p.Factory ◁ l.Object {
   p.Factory() { super(); } }
   p.Table table() { return new p.Table( new p.Buck() ); }
}
```

specifies that the method's return type is `p.Cell`; this is valid as that class is public. In this example a factory class, named `p.Factory`, is needed to create instances of `p.Table` because the table's constructor expects a bucket and since buckets are confined, they cannot be instantiated outside of their defining package.

This program does not preserve confinement as the body of the `p.Table.get()` method returns an instance of a confined class in violation of the widening rule. The breach can be exhibited by constructing a class `o.Breach` in package `o` which creates a new table and retrieves its bucket.

```
class o.Breach ◁ l.Object {
   l.Object main () { return new p.Factory().table().get(); }
}
```

The expression `new o.Breach().main()` eventually evaluates to to `new p.Buck()`, ex-

posing the confined class to code defined in another package. This example is not typable in the ConfinedFJ type system. The method `p.Table.get()` does not type-check because Rule T-Method requires the type of the expression returned by the method to be a safe subtype of the method's declared return type. The expression has the confined type `p.Buck` while the declared return type is the public type `p.Cell`.

In another prototypical breach of confinement, consider the following situation in which the confined class `p.Self` extends a `Broken` parent class that resides in package `o`. Assume further that the class inherits its parent's code for the `reveal()` method.

```
conf class p.Self ◁ o.Broken {
   p.Self() { super(); }
}

class p.Main ◁ l.Object {
   p.Main() { super(); }
   l.Object get() { return new p.Self().reveal(); }
}
```

Inspection of this code does not reveal any breach of confinement. But if we widen the scope of our analysis to the `o.Broken` class, we may see:

```
class o.Broken ◁ l.Object {
   o.Broken() { super(); }
   l.Object reveal() { return this; }
}
```

Invoking `reveal()` on an instance of `p.Self` will return a reference to the object itself. This does not type-check because the invocation of `reveal()` in `p.Main.get()` violates the Rule T-Invk (due to that the non-anonymous method `reveal()`, inherited from a public class `o.broken`, is invoked on an object of a confined type `p.Self`). The method `reveal()` cannot be declared anonymous as the method returns `this` directly.

Type Soundness

Zhao, Palsberg, and Vitek [75] presented a small-step operational semantics of ConfinedFJ, which is a computation-step relation $P \rightarrow P'$ on program states P, P'. They define that a program state *satisfies confinement* if every object is in the scope of its defining class. They proceed to prove the following type soundness result (for a version of ConfinedFJ without downcast).

Theorem (Confinement [75]) If P is well-typed, satisfies confinement, and $P \rightarrow^* P'$, then P' satisfies confinement.

The Confinement Theorem states that a well-typed program that initially satisfies confinement preserves confinement. Intuitively, this means that during the execution of a well-typed program, all the objects that are accessed within the body of a method are visible from the method's defining package. The only exception is for anonymous methods, as they may have access to `this` which can evaluate to an instance of a class confined in another package, and if this occurs the use of `this` is restricted to be a receiver object.

Confined types have none of the three drawbacks of whole-program static analysis: we can type check fragments of code well before the entire program is done, the type system enforces a discipline that can help make many types confined, and a change to a line of code only affects types locally.

Confinement Inference

Every type-correct Featherweight Java program can be transformed into a type-correct ConfinedFJ program by putting all the classes into the same package. Conversely, every type-correct ConfinedFJ program can be transformed into a type-correct Java program by removing all occurrences of the modifiers `conf` and `anon`. (The original version of Featherweight Java does not have packages.)

The modifiers `conf` and `anon` can help enforce more discipline than Java does. If we begin with a program in Featherweight Java extended with packages and would like to enforce the stricter discipline of ConfinedFJ, then we face what we call the *confinement inference* problem.

> **The confinement inference problem.** Given a Java program, find a subset of the package-scoped classes that we can make confined, and find a subset of the methods that we can make anonymous.

The confinement inference problem has a trivial solution: make no classes confined and make no method anonymous. In practice we may want the largest subsets we can get.

Grothoff, Palsberg, and Vitek [44] studied confinement inference for a variant of the confinement and anonymity rules in this chapter. They use a constraint-based program analysis to infer confinement and method anonymity. Their constraint-based analysis proceeds in two steps: (1) generate a system of constraints from program text and then (2) solve the constraint system. The constraints are of following six forms:

$$
\begin{array}{rcl}
A & ::= & \mathsf{not\text{-}anon}(\text{methodid}) \\
T & ::= & \mathsf{not\text{-}conf}(\text{classid}) \\
C & ::= & A \mid T \mid T \Rightarrow A \mid A \Rightarrow A \mid A \Rightarrow T \mid T \Rightarrow T
\end{array}
$$

A constraint not-anon(methodId) asserts that the method methodId is *not* anonymous; similarly, not-conf(classId) asserts that the class classId is *not* confined. The remaining four forms

of constraints denote logical implications. For example, not-anon(A.m()) ⇒ not-conf(C) is read "if method m in class A is not anonymous then class C will not be confined."

From each expression in a program, we generate one or more constraints. For example, for a type cast expression (C) e for which the static Java type of e is D, we generate the constraint not-conf(C) ⇒ not-conf(D), which represents the condition from the T-UCast rule that *conf*(D) ⇒ *conf*(C).

All the constraints are ground Horn clauses. The solution procedure computes the set of clauses not-conf(classId) that are either immediate facts or derivable via logical implication. This computation can be done in linear time [31] in the number of constraints, which, in turn, is linear in the size of the program.

A solution represents a set of classes that cannot be confined and a set of methods that are not anonymous. The complements of those sets represent a maximal solution to the confinement inference problem.

Grothoff, Palsberg, and Vitek [44] presented an implementation of their constraint-based analysis. They gathered a suite of forty-six thousand Java classes and analyzed them for confinement. The average time to analyze a class file is less than eight milliseconds. The results show that, without any change to the source, 24% of the package-scoped classes (exactly 3,804 classes or 8% of all classes) are confined. Furthermore, they found that by using generic container types, the number of confined types could be increased by close to one thousand additional classes. Finally, with appropriate tool support to tighten access modifiers, the number of confined classes can be well over 14,500 (or over 29% of all classes) for that same benchmark suite.

1.2.4 Related Work on Alias Control

The type-based approach in this chapter is one among many type-based approaches that solve the same or similar problems. For example, a popular approach is to use a notion of ownership type [2, 4, 5, 11, 12, 14, 20, 21, 28, 50, 56] for controlling aliasing. The basic idea

of ownership types is to use the concept of *domination* on the dynamic object graph. (In a graph with a starting vertex s, a vertex u dominates another vertex v if every path from s to v must pass through u.) In a dynamic object graph, we may have an object which we think of as *owning* several *representation* objects. The goal of ownership types is to ensure that the owner object dominates the representation objects. The dominance relation guarantees that the only way we can access a representation object is via the owner. An ownership type system has type rules that are quite different than the rules for confined types.

1.3 Type Qualifiers

In this section we will use types to allow programmers to easily specify and check desired properties of their applications. This is achieved by allowing programmers to introduce new *qualifiers* that refine existing types. For example, the type `nonzero int` is a refinement of the type `int` that intuitively denotes the subset of integers other than zero.

1.3.1 Background

Static type systems are useful for catching common programming errors early in the software development cycle. For example, type systems can ensure that an integer is never accidentally used as a string and that a function is always passed the right number and kinds of arguments. Unfortunately, language designers cannot anticipate all of the program errors that programmers will want to statically check, nor can they anticipate all of the practical ways in which such errors can be checked.

As a simple example, while most type systems in mainstream programming languages can distinguish integers from strings and ensure that each kind of data is used in appropriate ways, these type systems typically cannot distinguish positive from negative integers. Such an ability would enable stronger assurances about a program, for example that it never attempts to take the square root of a negative number. As another example, most type systems cannot distinguish between data that originated from one source and data that

originated from a different source within the program. Such an ability could be useful to track a form of *value flow*, for example to ensure that a string that was originally input from the user is treated as *tainted* and therefore given restricted capabilities (e.g., such a string should be disallowed as the format-string argument to C's `printf` function, since a bad format string can cause program crashes and worse).

Without static checking for these and other kinds of errors, programmers have little recourse. They can use `assert` statements, which catch errors but only as they occur in a running system. They can specify desired program properties in comments, which are useful documentation but need have no relation to the actual program behavior. In the worst case, programmers simply leave the desired program properties completely implicit, making these properties easy to misunderstand or forget entirely.

1.3.2 Static Analysis

Static analysis could be used to ensure desired program properties and thereby guarantee the absence of classes of program errors. Indeed, generic techniques exist for performing static analyses of programs (e.g., [49, 23]), which could be applied to the properties of interest to programmers. As with confinement, one standard approach is to compute a flow set for each expression e in the program, which conservatively over-approximates the possible values of e. However, instead of using class names as the elements of a flow set, each static analysis defines its own domain of *flow facts*.

For example, to track positive and negative integers, a static analysis could use a domain of signs [23], consisting of the three elements `+`, `0`, and `-` with the obvious interpretations. If the flow set computed for an expression e contains only the element `+`, then we can be sure that e will evaluate to a positive integer. In our format-string example, a static analysis could use a domain consisting of the elements `tainted` and `untainted`, respectively representing data that does and does not come from the user. If the flow set computed for an expression e contains only the element `untainted`, then we can be sure that e does not come from the user.

$$\begin{array}{lcl} \tau & ::= & \texttt{int} \mid \tau \to \tau \\ e & ::= & n \mid e_1 + e_2 \mid x \mid \lambda x : \tau . e \mid e_1 e_2 \end{array}$$

Figure 1.5: The syntax of the simply typed lambda calculus.

While this approach is general, it suffers from the drawbacks discussed in Chapter 1.2.2. First, whole-program analysis is typically required for precision, so errors are only caught once the entire program has been implemented. Second, the static analysis is *descriptive*, reporting the properties that are true of a given program, rather than *prescriptive*, providing a discipline to help programmers achieve the desired properties. Finally, the results of a static analysis can be sensitive to small changes in the program.

The type-based approach described next is less precise than some static analyses but has none of the above drawbacks.

1.3.3 A Type System for Qualifiers

We now develop a type system that supports programmer-defined type qualifiers. After a brief review of the simply typed lambda calculus, types are augmented with user-defined *tags* and language support for *tag checking*. A notion of subtyping for tagged types provides a natural form of type qualifiers. Finally, more expressiveness is achieved by allowing users to provide specialized typing rules for qualifier checking.

Simply Typed Lambda Calculus

We assume familiarity with the simply typed lambda calculus and briefly review here the portions that are relevant for the rest of the section. Many other sources contain fuller descriptions of the simply typed lambda calculus, for example the text by Pierce [65].

Figure 1.5 shows the syntax for the simply typed lambda calculus augmented with integers and integer addition. The metavariable τ ranges over types and e ranges over expressions. The syntax $\tau_1 \to \tau_2$ denotes the type of functions with argument type τ_1 and result type τ_2. The metavariable n ranges over integer constants and x ranges over variable names. The

$\boxed{\Gamma \vdash e \ : \ \tau}$

$$\Gamma \vdash n \ : \ \texttt{int} \qquad \text{(T-Int)}$$

$$\frac{\Gamma \vdash e_1 \ : \ \texttt{int} \qquad \Gamma \vdash e_2 \ : \ \texttt{int}}{\Gamma \vdash e_1 + e_2 \ : \ \texttt{int}} \qquad \text{(T-Plus)}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x \ : \ \tau} \qquad \text{(T-Var)}$$

$$\frac{\Gamma, x : \tau_1 \vdash e \ : \ \tau_2}{\Gamma \vdash \lambda x : \tau_1 . e \ : \ \tau_1 \rightarrow \tau_2} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash e_1 \ : \ \tau_2 \rightarrow \tau \qquad \Gamma \vdash e_2 \ : \ \tau_2}{\Gamma \vdash e_1 e_2 \ : \ \tau} \qquad \text{(T-App)}$$

Figure 1.6: Static typechecking for the simply typed lambda calculus.

syntax $\lambda x : \tau . e$ represents a function with formal parameter x (of type τ) and body e, and the syntax $e_1 e_2$ represents application of the function expression e_1 to the actual argument e_2.

Figure 1.6 presents static typechecking rules for the simply typed lambda calculus. The rules define a judgment of the form $\Gamma \vdash e \ : \ \tau$. The metavariable Γ ranges over *type environments*, which are finite mappings from variables to types. Informally, the judgment $\Gamma \vdash e \ : \ \tau$ says that expression e is *well typed* with type τ under the assumption that free variables in e have the types associated with them in Γ. The rules in Figure 1.6 are completely standard.

Static typechecking enforces a notion of well-formedness on programs at compile time, thereby preventing some common kinds of run-time errors. For example, the rules in Figure 1.6 ensure that a well-typed expression (with no free variables) will never attempt to add an integer to a function at run time. A type system's notion of well-formedness is formalized by a type soundness theorem, which specifies the properties of well-typed programs. Intuitively, type soundness for the simply typed lambda calculus says that the evaluation of well-typed expressions will not "get stuck," which happens when an operation is attempted with operand values of the wrong types.

$$\begin{array}{rcl} \tau & ::= & q\ \nu \\ \nu & ::= & \texttt{int} \mid \tau \to \tau \\ e & ::= & \cdots \mid \texttt{annot}(e, q) \mid \texttt{assert}(e, q) \end{array}$$

Figure 1.7: Adding user-defined tags to the syntax.

A type soundness theorem relies on a formalization of a language's evaluation semantics. There are many styles of formally specifying language semantics and of proving type soundness, and common practice today is well described by others [71, 65]. These topics are beyond the scope of this chapter.

Tag Checking

One way to allow programmers to easily extend their type system is to augment the syntax for types with a notion of programmer-defined *type tags* (or simply *tags*). The new syntax of types is shown in Figure 1.7. The metavariable q ranges over an infinite set of programmer-definable type tags. Each type is now augmented with a tag. For example, `positive int` could be a type, where `positive` is a programmer-defined tag denoting positive integers. Function types include a top-level tag as well as tags for the argument and result types.

In order for programmers to convey the intent of a type tag, the language is augmented with two new expression forms, as shown in Figure 1.7. Our presentation follows that of Foster *et al.* [38, 39]. The expression $\texttt{annot}(e, q)$ evaluates e and tags the resulting value with q. For example, if the expression e evaluates to a string input by the user, one can use the expression `annot(`e`, tainted)` to declare the intention to consider e's value as tainted [58, 66]. The expression $\texttt{assert}(e, q)$ evaluates e and checks that the resulting value is tagged with q. For example, the expression `assert(`e`, untainted)` ensures that e's value does not originate from the user and is therefore an appropriate format-string argument to `printf`. A failed `assert` causes the program to terminate erroneously.

Just as our base type system in Figure 1.6 statically tracks the type of each expression, so does our augmented type system, using the augmented syntax of types. The rules are shown in Figure 1.8. For simplicity, the rules are set up so that each run-time value created during the program's execution will have exactly one tag (a conceptually untagged value can

$\boxed{\Gamma \vdash e \ : \ \nu}$

$$\Gamma \vdash n \ : \ \texttt{int} \qquad \text{(Q-Int)}$$

$$\frac{\Gamma \vdash e_1 \ : \ q_1 \ \texttt{int} \qquad \Gamma \vdash e_2 \ : \ q_2 \ \texttt{int}}{\Gamma \vdash e_1 + e_2 \ : \ \texttt{int}} \qquad \text{(Q-Plus)}$$

$$\frac{\Gamma, x : \tau_1 \vdash e \ : \ \tau_2}{\Gamma \vdash \lambda x : \tau_1 . e \ : \ \tau_1 \rightarrow \tau_2} \qquad \text{(Q-Abs)}$$

$\boxed{\Gamma \vdash e \ : \ \tau}$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x \ : \ \tau} \qquad \text{(Q-Var)}$$

$$\frac{\Gamma \vdash e_1 \ : \ \tau_2 \rightarrow \tau \qquad \Gamma \vdash e_2 \ : \ \tau_2}{\Gamma \vdash e_1 e_2 \ : \ \tau} \qquad \text{(Q-App)}$$

$$\frac{\Gamma \vdash e \ : \ \nu}{\Gamma \vdash \texttt{annot}(e, q) \ : \ q \ \nu} \qquad \text{(Q-Annot)}$$

$$\frac{\Gamma \vdash e \ : \ q \ \nu}{\Gamma \vdash \texttt{assert}(e, q) \ : \ q \ \nu} \qquad \text{(Q-Assert)}$$

Figure 1.8: Adding user-defined tags to the type system.

be modeled by tagging it with a distinguished `notag` tag). This invariant is achieved via two interrelated typing judgments. The judgment $\Gamma \vdash e \ : \ \nu$ determines an untagged type for a given expression. This judgment is only defined for *constructor expressions*, which are expressions that dynamically create new values. The judgment $\Gamma \vdash e \ : \ \tau$ is the top-level typechecking judgment. It is defined for all other kinds of expressions. The Q-Annot rule provides a bridge between the two judgments, requiring each constructor expression to be tagged in order to be given a complete type τ.

Intuitively, the type system conservatively ensures that if $\Gamma \vdash e \ : \ q\ \nu$ holds then the value of e at run time will be tagged with q. The rules for $\texttt{annot}(e, q)$ and $\texttt{assert}(e, q)$ are straightforward: Q-Annot includes q as the tag on the type of e, while Q-Assert requires that e's type already includes the tag q. The rest of the rules are unchanged from the original simply typed lambda calculus, except that the premises of Q-Plus allow for the tags on the types of the operands. Nonetheless, these unchanged rules have exactly the desired effect. For example, Q-App requires the actual argument's type in a function application to match the formal argument type, thereby ensuring that the function only ever receives values tagged with the expected tag.

Together the rules in Figure 1.8 provide a simple form of value-flow analysis, statically ensuring that values of a given tag will flow at run time only to places where values of that tag are expected. For example, a programmer can define a square-root function of the form

$$\lambda x : \texttt{positive int}.e$$

and the type system guarantees that only values explicitly tagged as `positive` will be passed to the function. As another example, the programmer can statically detect possible division-by-zero errors by replacing each divisor expression e (assuming our language included integer division) with the expression $\texttt{assert}(e, \texttt{nonzero})$. Finally, the type of the following function, `tainted int`→`untainted int`, ensures that although the function accepts tainted data as an argument, this data does not flow to the return value:

$$\lambda x : \texttt{tainted int}.\texttt{annot}(0, \texttt{untainted})$$

On the other hand, the following function, which returns the given tainted argument, is forced to record this fact in its type, `tainted int`→`tainted int`:

$$\lambda x : \texttt{tainted int}.x$$

Type Soundness The notion of type soundness in the presence of tags is a natural extension of that for the simply typed lambda calculus. Type soundness still ensures that well-typed expressions won't get stuck, but the notion of stuckness now includes failed `assert`s. This definition of stuckness formalizes the idea that tagged values will only flow where they are expected. Type soundness can be proven using standard techniques.

Tag Inference It is possible to consider *tag inference* for our language. Constructor expressions are no longer explicitly annotated via `annot`, and formal argument types no longer include tags. Tag inference automatically determines the tag of each constructor expression and the tags on each formal argument, or determines that the program cannot be typed. Programmers still must employ `assert` explicitly in order to specify constraints on where values of particular tags are expected.

As with confinement inference, a constraint-based program analysis can be used for tag inference. Conceptually, each subexpression in the program is given its own *tag variable*, and the analysis then generates equality constraints based on each kind of expression. For example, in a function application, the tag of the actual argument is constrained to match the tag of the formal argument type. The simple equality constraints generated by tag inference can be solved in linear time [64, 69]. Further, if the constraints have a solution then there exists a *principal solution*, which is more general than every other solution. Intuitively, this is the solution that produces the largest number of tags.

For example, consider the following function:

$$\lambda x : \texttt{int}.\lambda y : \texttt{int}.\texttt{assert}(x, \texttt{tainted})$$

One possible typing for the function gives both x and y the type `tainted int`. However,

a more precise typing gives y's type a fresh tag q_y, since the function's constraints do not require it to have tag `tainted`. This new typing encodes that fact, as well as the fact that x and y flow to disjoint places in the program. Finally, the following program generates constraints that have no solution, since `x` is required to be both `tainted` and `untainted`:

$$(\lambda x : \texttt{int}.\texttt{assert}(x, \texttt{tainted}))\ \texttt{assert}(e, \texttt{untainted})$$

Qualifier Checking

While the type system in the previous subsection allows programmers to specify and check new properties of interest via tags, its expressiveness is limited by the fact that tags are completely uninterpreted. For example, the type system does not "know" the intent of tags like `positive`, `nonzero`, `tainted`, and `untainted`; it only knows that these tags are not equivalent to one another. However, tags often have natural relationships to one another. For example, intuitively it should be safe to pass a `positive int` where a `nonzero int` is expected, since a positive integer is also nonzero. Similarly, we may want to allow untainted data to be passed where tainted data is expected, since allowing that cannot cause tainted data to be improperly used. The type system of the previous section does not permit such flexibility.

Foster *et al.* observed that this expressiveness can be naturally achieved by allowing programmers to specify a partial order $\sqsubseteq$ on type tags [38, 39]. Intuitively, if $q_1 \sqsubseteq q_2$, then q_1 denotes a stronger constraint than q_2. The programmer can now declare `positive` $\sqsubseteq$ `nonzero` and similarly `untainted` $\sqsubseteq$ `tainted`, where `untainted` denotes the set of values that are definitely untainted and `tainted` now denotes the set of values that are *possibly* tainted. The programmer-defined partial order naturally induces a subtyping relation among tagged types. For example, given the above partial order, `positive int` would be considered a subtype of `nonzero int`, which therefore allows a value of the former type to be passed where a value of the latter type is expected.

With this added expressiveness, type tags can be considered full-fledged *type qualifiers*. For example, a canonical example of a type qualifier is C's `const` annotation, which indicates

$\boxed{\tau \leq \tau'}$

$$\frac{q \sqsubseteq q'}{q\ \texttt{int} \leq q'\ \texttt{int}} \quad \text{(S-Int)}$$

$$\frac{q \sqsubseteq q' \qquad \tau_1' \leq \tau_1 \qquad \tau_2 \leq \tau_2'}{q(\tau_1 \to \tau_1') \leq q'(\tau_2 \to \tau_2')} \quad \text{(S-Fun)}$$

$\boxed{\Gamma \vdash e\ :\ \tau}$

$$\frac{\Gamma \vdash e\ :\ \tau' \qquad \tau' \leq \tau}{\Gamma \vdash e\ :\ \tau} \quad \text{(Q-Sub)}$$

Figure 1.9: Adding subtyping to the type system.

that the associated value can be initialized but not later updated. C allows a value of type `int*` to be passed where a `(const int)*` is expected. This is safe because it simply imposes an extra constraint on the given pointer value, namely that its contents are never updated. On the other hand, a value of type `(const int)*` cannot safely be passed where an `int*` is expected, since this would allow the pointer value's `const`ness to be forgotten, allowing its contents to be modified. Another useful example qualifier is `nonnull` for pointers, whereby it is safe to pass a `nonnull` pointer where an arbitrary pointer is expected, but not vice versa.

Figure 1.9 shows the extension of the rules in the previous subsection to support qualifiers, adapted from [39]. Q-Sub is a *subsumption* rule, which allows an expression's type to be promoted to any supertype. The subtyping relation $\leq$ depends on the partial order $\sqsubseteq$ among qualifiers in a straightforward way. As usual, subtyping is *contravariant* on function argument types for soundness [16].

As an example of this type system in action, consider an expression e of type `positive int`. Assuming that the programmer specifies `positive` $\sqsubseteq$ `nonzero`, then by S-Int we have `positive int` $\leq$ `nonzero int` and by Q-Sub e also has type `nonzero int`. Therefore, by the `Q-App` rule from Figure 1.8, e may be passed to a function expecting an argument of type `nonzero int`.

As an aside, the addition of subtyping makes our formal system expressive enough to encode multiple qualifiers per type. For example, to encode a type like `untainted positive int`, one can define a new qualifier `untainted_positive` along with the partial order `untainted_positive` $\sqsubseteq$ `untainted` and `untainted_positive` $\sqsubseteq$ `positive`. Then the subtyping and subsumption rules allow an `untainted_positive` value to be treated as being both `untainted` and `positive`, as desired.

As before, type soundness says that the type system guarantees that all `assert`s will succeed at run time, where the run-time assertion check now requires a value's associated qualifier to be "less than" the specified qualifier, according to the declared partial order. The type soundness proof again uses standard techniques. It is also possible to generalize tag inference to support *qualifier inference*. The approach is similar to that described above, although the generated constraints are now *subtype constraints* instead of equality constraints.

Foster's thesis discusses both type soundness and qualifier inference in detail [37]. It also discusses CQual, an implementation of programmer-defined type qualifiers that adapts the described theory to the C language. CQual has been used successfully for a variety of applications, including inference of `const`ness [39], detection of format-string vulnerabilities [66], detection of user/kernel pointer errors [47], validation of the placement of authorization hooks in the Linux kernel [74], and the removal of sensitive information from crash reports [15].

Qualifier-Specific Typing Rules

The $\sqsubseteq$ partial order allows programmers to specify more information about each qualifier, making the overall type system more flexible. However, most of the intent of a qualifier must still be conveyed indirectly via `annot`s, which is tedious and error prone. For example, the programmer must use `annot` to explicitly annotate each constructor expression that evaluates to a positive integer as being `positive`, or else it will not be considered as such by the type system. Therefore, the programmer has the burden of manually figuring out which expressions are positive and which are not. Further, if the programmer accidentally annotates an expression like `-34 + 5` as `positive`, the type system will happily allow this

```
qualifier positive(int Expr E)
  case E of
      decl int Const C:
        C, where C > 0
    | decl int Expr E1, E2:
        E1 + E2, where positive(E1) && positive(E2)
```

Figure 1.10: A programming discipline for `positive` in Clarity.

expression to be passed to a square-root function expecting a `positive int`, even though that will likely cause a run-time error.

Qualifier inference avoids the need for explicit annotations using `annot`. However, qualifier inference simply determines which expressions must be treated as `positive` in order to satisfy a program's `assert`s. There is no guarantee that these expressions actually evaluate to positive integers, and many expressions that do evaluate to positive integers will not be found to be `positive` by the inferencer.

To address the burden and fragility of qualifier annotations, we consider an alternate approach to expressing a qualifier's intent. Instead of relying on program annotations, we require qualifier designers to specify a *programming discipline* for each qualifier, which indicates when an expression may be given that qualifier. For example, a programming discipline for `positive` might say that all positive constants can be considered `positive`, and that an expression of the form $e_1 + e_2$ can be considered `positive` if each operand expression can itself be considered `positive` according to the discipline. In this way, the discipline declaratively expresses the fact that `34 + 5` can be considered `positive` while `-34 + 5` cannot.

The approach described is used by the Clarity framework for programmer-defined type qualifiers in C [17]. Clarity provides a declarative language for specifying programming disciplines. For example, Figure 1.10 shows how the discipline informally described above for `positive` would be specified in Clarity. The figure declares a new qualifier named `positive`, which refines the type `int`. It then uses pattern matching to specify two ways in which an expression `E` can be given the qualifier `positive`. The Clarity framework includes

an *extensible typechecker*, which employs user-defined disciplines to automatically typecheck programs.

Formally, consider the type system consisting of the rules in Figures 1.8 and 1.9. We remove all the rules of the form $\Gamma \vdash e \ : \ \nu$, which perform typechecking on constructor expressions, and we remove the `annot` expression form along with its typechecking rule Q-Annot. When a programmer introduces a new qualifier, she must also augment the type system with new inference rules indicating the conditions under which each constructor expression may be given this qualifier. For example, the rules in Figure 1.10 are formally represented by adding the following two rules to the type system:

$$\frac{n > 0}{\Gamma \vdash n \ : \ \texttt{positive int}} \qquad \text{(P-Int)}$$

$$\frac{\Gamma \vdash e_1 \ : \ \texttt{positive int} \qquad \Gamma \vdash e_2 \ : \ \texttt{positive int}}{\Gamma \vdash e_1 + e_2 \ : \ \texttt{positive int}} \qquad \text{(P-Plus)}$$

Assuming that the programmer also declares `positive` $\sqsubseteq$ `nonzero`, the subtyping and subsumption rules in Figure 1.9 allow the above rules to be used to give an expression the qualifier `nonzero` as well.

Not all qualifiers have natural rules associated with them. For example, the programming disciplines associated with qualifiers like `tainted` and `untainted` could be program-dependent and/or quite complicated. Therefore, in practice both the Clarity and CQual approaches are useful.

Type Soundness A type soundness theorem analogous to that for traditional type qualifiers, which guarantees that `assert`s succeed at run time, can be proven in this setting. In addition, it is possible to prove a stronger notion of type soundness. Clarity allows the programmer to optionally specify the set of values associated with a particular qualifier. For example, the programmer could associate the set of positive integers with the `positive` qualifier. Given this information, type soundness says that a well-typed expression with qualifier `positive` will in fact evaluate to a member of the specified set.

To ensure this form of type soundness, Clarity generates one *proof obligation* per programmer-defined rule. For example, the second rule for `positive` above requires proving that the sum of two integers greater than zero is also an integer greater than zero. Clarity discharges proof obligations automatically using off-the-shelf decision procedures [29], but in general these may need to be manually proven by the qualifier designer.

This form of type soundness serves to validate the programmer-defined rules. For example, if the second rule for `positive` above were erroneously defined for subtraction rather than addition, then the error would be caught because the associated proof obligation is not valid: the difference of two positive integers is not necessarily positive. In this way, programmers obtain a measure of confidence that their qualifiers and associated inference rules are behaving as intended.

Qualifier Inference Qualifier inference is also possible in this setting and is implemented in Clarity, allowing the qualifiers for variables to be inferred rather than declared by the programmer. Similar to qualifier inference in the previous subsection, a set of subtype constraints is generated and solved. However, handling programmer-defined inference rules requires a form of *conditional* subtype constraints to be solved [18].

1.3.4 Related Work on Type Refinements

Work on *refinement types* for the ML language allows programmers to create subtypes of datatype definitions [41], each denoting a subset of the values of the datatype. For example, a standard list datatype could be refined to define a type of non-empty lists. The language for specifying these refinements is analogous to the language for programmer-defined inference rules in Clarity.

Other work has shown how to make refinement types and type qualifiers *flow sensitive* [26, 40, 51, 27], which allows the refinement of an expression to change over time. For example, a file pointer could have the qualifier `closed` upon creation and the qualifier `open` after it has been opened. In this way, type refinements can be used to track *temporal protocols*, for

example that a file must be opened before it can be read or written.

Finally, others have explored type refinements through the notion of *dependent types* [52], in which types can depend on program expressions. An instance of this approach is Dependent ML [72, 73], which allows types to be refined through their dependence on linear arithmetic expressions. For example, the type `int list(5)` represents integer lists of length 5, and a function that adds an element to an integer list would be declared to have the argument type `int list(n)` for some integer `n` and to return a value of type `int list(n+1)`. These kinds of refinements are targeted at qualitatively different kinds of program properties from those targeted by type qualifiers.

References

[1] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 311–330. ACM Press, 2002.

[2] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Appplications (OOPSLA)*, pages 311–330, November 2002.

[3] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)*, pages 324–341, San Jose, CA, 1996. *SIGPLAN Notices* 31(10).

[4] Anindya Banerjee and David A. Naumann. Representation independence, confinement and access control. In *Proceedings of POPL'02, SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 166–177, 2002.

[5] Mike Barnett, Robert DeLine, Manuel Fäahndrich, K. Rustan M. Leino, , and Wolfram Schulte. Verification of object-oriented programs with invariants. In *Fifth Workshop on Formal Techniques for Java-like Programs*, 2003.

[6] Bruno Blanchet. Escape analysis for object oriented languages. application to Java. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, pages 20–34, Denver, CO, October 1999. ACM Press.

[7] Bruno Blanchet. Escape analysis for Java: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 25(6):713–775, 2003.

[8] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in Java. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, pages 35–46, Denver, CO, October 1999. ACM Press.

[9] Boris Bokowski and Jan Vitek. Confined types. In *Proceedings of the Fourteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)*, pages 82–96, Denver, CO, 1999.

[10] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 211–230. ACM Press, 2002.

[11] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Appplications (OOPSLA)*, pages 211–230, November 2002.

[12] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, and Martin Rinard. Ownership types for safe region-based memory management in real-time Java. In *ACM Conference on Programming Language Design and Implementation*, pages 324–337, June 2003.

[13] John Boyland. Alias burying: Unique variables without destructive reads. *Softw. Pract. Exper.*, 31(6):533–553, 2001.

[14] John Boyland. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience*, 31(6):533–553, 2001.

[15] Pete Broadwell, Matt Harren, and Naveen Sastry. Scrash: A System for Generating Secure Crash Information. In *USENIX Security Symposium*, 2003.

[16] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*,

76(2/3):138–164, February 1988.

[17] Brian Chin, Shane Markstrum, and Todd Millstein. Semantic type qualifiers. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–95, New York, NY, USA, 2005. ACM Press.

[18] Brian Chin, Shane Markstrum, Todd Millstein, and Jens Pal sberg. Inference of user-defined type qualifiers and qualifier rules. In *European Symposium on Programming*, 2006.

[19] Dave Clarke, Michael Richmond, and James Noble. Saving the world from bad Beans: Deployment-time confinement checking. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Appplications (OOPSLA)*, pages 374–387, Anaheim, CA, November 2003.

[20] David Clarke. *Object Ownership and Containment*. PhD thesis, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia, 2001.

[21] David Clarke and Tobias Wrigstad. External uniqueness. In *10th Workshop on Foundations of Object-Oriented Languages (FOOL)*, New Orleans, LA, January 2003.

[22] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 48–64. ACM Press, 1998.

[23] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[24] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: path-sensitive program verification in polynomial time. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 57–68, New York, NY, USA, 2002. ACM Press.

[25] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In W. Olthoff, editor, *Proceedings of the Ninth European Conference on Object-Oriented Programming (ECOOP'95)*, pages 77–101, Aarhus, Denmark, August 1995. Springer-Verlag.

[26] Robert DeLine and Manuel Fahndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 59–69. ACM Press, 2001.

[27] Robert DeLine and Manuel Fahndrich. Typestates for objects. In *Proceedings of the 2004 European Conference on Object-Oriented Programming*, LNCS 3086, Oslo, Norway, June 2004. Springer-Verlag.

[28] David Detlefs, K. Rustan M. Leino, and Greg Nelson. Wrestling with rep exposure. Technical report, Digital Equipment Corporation Systems Research Center, 1996.

[29] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[30] Alain Deutsch. Semantic models and abstract interpretation techniques for inductive data structures and pointers. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 226–229, La Jolla, California, June 21–23, 1995.

[31] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming*, 1(3):267–84, October 1984.

[32] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[33] David Evans. Static detection of dynamic memory errors. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 44–53, New York, NY, USA, 1996. ACM Press.

[34] Manuel Fahndrich and K. Rustan M. Leino. Declaring and checking non-null types in

an object-oriented language. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 302–312. ACM Press, 2003.

[35] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 219–232. ACM Press, 2000.

[36] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 338–349. ACM Press, 2003.

[37] Jeffrey S. Foster. *Type Qualifiers: Lightweight Specifications to Improve Software Quality*. Ph.D. dissertation, University of California, Berkeley, 2002.

[38] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A Theory of Type Qualifiers. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 192–203, Atlanta, Georgia, May 1999.

[39] Jeffrey S. Foster, Robert Johnson, John Kodumal, and Alex Aiken. Flow-insensitive type qualifiers. *ACM Transactions on Programming Languages and Systems*, 28(6), November 2006.

[40] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 1–12. ACM Press, 2002.

[41] Tim Freeman and Frank Pfenning. Refinement types for ML. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 268–277, New York, NY, USA, 1991. ACM Press.

[42] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[43] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 282–293. ACM Press, 2002.

[44] Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. *ACM Transactions on Programming Languages and Systems*. To appear. Preliminary version in Proceedings of OOPSLA'01, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, pages 241–253, Tampa Bay, Florida, October 2001.

[45] Nevin Heintze. Control-flow analysis and type systems. In *Proceedings of SAS'95, International Static Analysis Symposium*, pages 189–206. Springer-Verlag (*LNCS* 983), Glasgow, Scotland, September 1995.

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

[47] Rob Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *Proceedings of the 13th USENIX Security Symposium*, pages 119–134, 2004.

[48] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[49] Gary A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206, Boston, Massachusetts, October 1973.

[50] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In *Proceedings of ECOOP'04, 16th European Conference on Object-Oriented Programming*, pages 491–516, 2004.

[51] Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. In *Proceedings of the eighth ACM SIGPLAN international conference on*

Functional programming, pages 213–225. ACM Press, 2003.
[52] Per Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North-Holland.
[53] Microsoft. Microsoft Visual C#. http://msdn.microsoft.com/vcsharp.
[54] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. Talx86: A realistic typed assembly language. Presented at 1999 ACM Workshop on Compiler Support for System Software, May 1999.
[55] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In *Proceedings of POPL'98, 25th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 85–97, 1998.
[56] Peter Müller and Arnd Poetzsch-Heffter. Universes: A type system for controlling representation exposure. In A. Poetzsch-Heffter and J. Meyer, editors, *Programming Languages and Fundamentals of Programming*. Fernuniversität Hagen, 1999.
[57] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 128–139. ACM Press, 2002.
[58] Peter Ørbæk and Jens Palsberg. Trust in the λ-calculus. *Journal of Functional Programming*, 7(6):557–591, November 1997. Preliminary version in Proceedings of SAS'95, International Static Analysis Symposium, Springer-Verlag (*LNCS* 983), pages 314–330, Glasgow, Scotland, September 1995.
[59] Jens Palsberg. Equality-based flow analysis versus recursive types. *ACM Transactions on Programming Languages and Systems*, 20(6):1251–1264, 1998.
[60] Jens Palsberg. Type-based analysis and applications. In *Proceedings of PASTE'01, ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 20–27, Snowbird, Utah, June 2001. Invited paper.
[61] Jens Palsberg and Patrick M. O'Keefe. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems*, 17(4):576–599, July 1995. Preliminary version in Proceedings of POPL'95, 22nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages, pages 367–378, San Francisco, California, January 1995.
[62] Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. *Journal of Functional Programming*, 11(3):263–317, May 2001. Preliminary version in Proceedings of POPL'98, 25th Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages, pages 197–208, San Diego, California, January 1998.
[63] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *Proceedings of OOPSLA'91, ACM SIGPLAN Sixth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 146–161, Phoenix, Arizona, October 1991.
[64] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.
[65] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
[66] 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.
[67] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of OOPSLA'00, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 281–293, Minneapolis, Minnesota, October 2000.
[68] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-

calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201. ACM Press, 1994.

[69] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamentae Informaticae*, X:115–122, 1987.

[70] John Whaley and Monica Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of PLDI'04, ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2004.

[71] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.

[72] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, Montreal, June 1998.

[73] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, January 1999.

[74] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using cqual for static analysis of authorization hook placement. In Dan Boneh, editor, *USENIX Security Symposium*, pages 33–48. USENIX, 2002.

[75] Tian Zhao, Jens Palsberg, and Jan Vitek. Type-based confinement. *Journal of Functional Programming*, 16(1):83–128, 2006. Preliminary version, entitled "Lightweight confinement for Featherweight Java", in Proceedings of OOPSLA'03, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, pages 135-148, Anaheim, California, October 2003.