Inference and Checking of Context-Sensitive Pluggable Types

Ana Milanova Wei Huang
Rensselaer Polytechnic Institute
110 8th Street, Troy NY
{milanova,huangw5}@cs.rpi.edu

ABSTRACT

Pluggable types can help find bugs such as null-pointer dereference or unwanted mutation (or they can prove the absence of such bugs). Unfortunately, pluggable types require annotations, which imposes a burden on programmers.

We propose a framework for specifying, *inferring* and checking of context-sensitive pluggable types. Programmers can use the framework to plug existing context-sensitive type systems (e.g., Immutability, Ownership) as well as to build new systems.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming

Keywords

type inference

1. INTRODUCTION

A pluggable type system enhances a traditional type system; examples include ReIm, a type system for reference immutability [10], Ownership Types [2], Universe Types [5], Non-null Types [6], and many others. Recently, pluggable type systems for concurrency [14] and energy efficiency [12] have been proposed. Pluggable types have numerous software engineering benefits. First, a pluggable type system can find bugs, or verify the absence of bugs. Second, it can reveal or enforce important program properties.

Recent papers report encouraging results on the usability of pluggable types. Dietl et al. report that pluggable types are easy to use productively, even by novice programmers [3]. Gordon et al. describe a system for C#, which has been in active use by a team at Microsoft [7].

Just as with a traditional type system, a pluggable type system requires annotations in the source code, and many pluggable type systems require a sizable amount of annotations. We believe that it is important to develop type inference techniques in order to further advance adoption of pluggable types.

2. THE NEW IDEA

Our new idea is that a large class of pluggable type systems can be *specified, inferred, and checked within a unified framework.* Programmers supply a few parameters (to be explained shortly) which instantiate the framework's typing rules into concrete typing rules for a specific system. The framework then takes an unannotated or a partially annotated program and fills in the remaining types. All of the above-mentioned type systems, as well as others, can be instantiated in the framework. Programmers can use the framework to (1) infer and plug existing type systems, and (2) build new type systems.

One key novelty of our system is the use of *viewpoint adaptation*, a concept from ownership types [5, 2, 14], to encode context sensitivity. We generalize traditional viewpoint adaptation to allow different kinds of context sensitivity such as *object sensitivity* and *call-site context sensitivity.* Another novelty is an efficient (low-polynomial) inference approach.

Below, we present the unified typing rules and instantiate these rules for two type system, ReIm, and Universe Types. We continue to present the unified inference approach.

2.1 Unified Typing Rules

Figure 1 shows the unified typing rules over a named-form Java syntax. For brevity, we assume that methods have exactly one parameter. In contrast to a formalization of pure Java, types of variables have two orthogonal components: pluggable type qualifier q and Java class type C as in q C x. The pluggable type system is *orthogonal* (i.e., independent) to the Java type system, which allows us to specify typing rules over type qualifiers q alone.

The framework is instantiated to a specific type system by defining three framework parameters: (1) the set of type qualifiers U with the corresponding subtyping hierarchy; qualifiers q in Figure 1 are members of U, (2) the viewpoint adaptation function (described below), and (3) type-system-specific constraints $\mathcal{B}$, enforced in addition to the standard subtyping constraints.

Viewpoint Adaptation. Many pluggable type systems (e.g., reference immutability, ownership) are context-sensitive. Our framework encodes context-sensitivity by using viewpoint adaptation. Traditional viewpoint adaptation in ownership types adapts the type of a field, formal parameter, or return, from the point of view of the *receiver* at the corresponding field access or method call; in other words,

$$\overset{(\textsc{tnew})}{\frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad q <: q_{\mathsf{x}} \quad \mathcal{B}_{(\textsc{tnew})}(q_{\mathsf{x}}, q)}{\Gamma \vdash \mathsf{x} = \mathsf{new}\ q\ \mathsf{C}}}$$

$$\overset{(\textsc{tassign})}{\frac{\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad q_{\mathsf{y}} <: q_{\mathsf{x}} \quad \mathcal{B}_{(\textsc{tassign})}(q_{\mathsf{x}}, q_{\mathsf{y}})}{\Gamma \vdash \mathsf{x} = \mathsf{y}}}$$

$$\overset{(\textsc{twrite})}{\frac{\begin{array}{c}\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \mathit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \\ q_{\mathsf{y}} \ <:\ a \rhd q_{\mathsf{f}} \quad \mathcal{B}_{(\textsc{twrite})}(q_{\mathsf{x}}, q_{\mathsf{f}}, q_{\mathsf{y}})\end{array}}{\Gamma \vdash \mathsf{x.f} = \mathsf{y}}}$$

$$\overset{(\textsc{tread})}{\frac{\begin{array}{c}\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \mathit{typeof}(\mathsf{f}) = q_{\mathsf{f}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \\ a \rhd q_{\mathsf{f}} \ <:\ q_{\mathsf{x}} \quad \mathcal{B}_{(\textsc{tread})}(q_{\mathsf{y}}, q_{\mathsf{f}}, q_{\mathsf{x}})\end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{y.f}}}$$

$$\overset{(\textsc{tcall})}{\frac{\begin{array}{c}\Gamma(\mathsf{x}) = q_{\mathsf{x}} \quad \Gamma(\mathsf{y}) = q_{\mathsf{y}} \quad \Gamma(\mathsf{z}) = q_{\mathsf{z}} \quad \mathit{typeof}(\mathsf{m}) = q_{\mathsf{this}}, q_p \rightarrow q_{\mathsf{ret}} \\ q_{\mathsf{y}} \ <:\ a \rhd q_{\mathsf{this}} \quad q_{\mathsf{z}} \ <:\ a \rhd q_p \\ a \rhd q_{\mathsf{ret}} \ <:\ q_{\mathsf{x}} \quad \mathcal{B}_{(\textsc{tcall})}(\mathsf{m}, q_{\mathsf{y}}, q_{\mathsf{x}})\end{array}}{\Gamma \vdash \mathsf{x} = \mathsf{y.m(z)}}}$$

Figure 1: Unified typing rules. Qualifiers q are taken from the universe of type qualifiers. Function *typeof* retrieves the declared qualifiers of fields and methods. Γ is a type environment that maps variables to their qualifiers. $\mathcal{B}$ sets contain additional constraints imposed by a specific type system. a at rules (TREAD), (TWRITE) and (TCALL) is the context of adaptation, a framework parameter.

traditional viewpoint adaptation adapts the type of a field, parameter or return to the "context" of the receiver. Viewpoint adaptation of a type q' from the point of view of another type q, results in the adapted type q''.

For example, the type of x.f is not just the declared type of field f — it is the type of f adapted from the point of view of x. As a concrete example, in Universe Types [5], if the declared type of f is peer, which denotes that the x object and its f field have the same owner, then the ownership type of x.f is not necessarily peer, but is the same as the type of x, because the relationship of x to the current object this is the same as the relationship of x.f to this.

Our framework generalizes traditional viewpoint adaptation: it allows for *variation* in the context of adaptation — one can adapt from the point of view of the receiver (as in ownership types), or one can adapt from the point of view of another variable. This allows us to express different kinds of context sensitivity.

Typing Rules. Let us look again at Figure 1. In rules (TWRITE) and (TREAD) framework parameters a and $\rhd$ handle viewpoint adaptation at field access. a is the *context of adaptation*. $a \rhd q_{\mathsf{f}}$ adapts the type of field f, namely q_{f}, from the point of view of a. Consider rule (TWRITE) x.f = y in Figure 1. The type of field f is adapted from the point of view of a and the expected subtyping constraint is created: $q_{\mathsf{y}} \ <:\ a \rhd q_{\mathsf{f}}$. a is usually the type of the receiver q_{x} (although different systems may use a different type as context of adaptation); when a is q_{x} the rule becomes $q_{\mathsf{y}} \ <:\ q_{\mathsf{x}} \rhd q_{\mathsf{f}}$.

Analogously, in (TCALL), framework parameters a and $\rhd$ handle viewpoint adaptation at method calls. a is the *context of adaptation* again. It can be instantiated to the type of the receiver q_{y}, which essentially amounts to object sensitivity [11], or to the type of the left-hand-side of the call assignment q_{x}, which amounts to call-site context sensitivity. $a \rhd q$ adapts formal parameter/return type q from the point of view of a. Rule (TCALL) creates the expected subtyping constraints.

2.2 Instantiations

To make the typing rules in Section 2.1 more concrete, we instantiate them into two different type systems: ReIm and Universe Types.

2.2.1 ReIm

ReIm [10] enforces the property that the state of an object, including its transitively reachable state, cannot be mutated through an immutable reference. ReIm is similar, but not identical to Javari [13]. In order to instantiate ReIm in the framework, we need to specify the framework parameters discussed in Section 2.1.

Type Qualifiers and Subtyping Hierarchy. We first instantiate the set of type qualifiers. There are three source-level type qualifiers in ReIm:

$$U = \{\mathsf{readonly}, \mathsf{polyread}, \mathsf{mutable}\}$$

- mutable: A mutable reference can be used to mutate the referenced object; this is the implicit and only option in standard object-oriented languages.
- readonly: A readonly reference x cannot be used to mutate the referenced object nor anything it references.
- polyread: A polyread reference x cannot be used to mutate the referenced object. Programmers should use polyread when the reference is readonly in the scope of the enclosing method, but depends on the context of the caller of the method.

Detailed explanations and examples of these qualifiers can be found in [10]. The subtyping hierarchy between the qualifiers is

$$\mathsf{mutable} <: \mathsf{polyread} <: \mathsf{readonly}$$

Viewpoint Adaptation Function. Next, we instantiate the viewpoint adaptation function:

$$\begin{array}{rcl} \_ \rhd \mathsf{mutable} & = & \mathsf{mutable} \\ \_ \rhd \mathsf{readonly} & = & \mathsf{readonly} \\ q \rhd \mathsf{polyread} & = & q \end{array}$$

_ denotes a don't care value. The context of adaptation at field access is *the type of the receiver* (x at field write x.f=y and y at field read x=y.f). A readonly (mutable) field remains readonly (mutable), regardless of the type of the receiver. A polyread field takes the type of the receiver after adaptation. For example, the type of field access y.f, where y is readonly and f is polyread, is the adapted type readonly $\rhd$ polyread = readonly.

Viewpoint adaptation at method call uses *the type of the left-hand-side of the call assignment* as context of adaptation. For example, the context of adaptation at the method call x = y.m(z) is the type of x. This abstracts call-site context sensitivity which is the kind of context sensitivity needed in ReIm as explained in [10].

Additional Constraints. Finally, we instantiate the additional constraints $\mathcal{B}$. All $\mathcal{B}$ sets are empty except for rule (TWRITE) where we have $\mathcal{B}_{(\text{TWRITE})}(q_r, q_f, q_o) = \{q_r = \mathsf{mutable}\}$; this constraint enforces that the receiver x at (TWRITE) x.f= y is mutable.

2.2.2 *Universe Types*

Universe Types [5] is a lightweight ownership type system that optionally enforces the *owner-as-modifier* encapsulation discipline. Informally, this means that an object can be modified only by its owner and by its peers, i.e., objects that have the same owner.

Type Qualifiers and Subtyping Hierarchy. There are three source-level type qualifiers in Universe Types:

$$U = \{\mathsf{peer}, \mathsf{rep}, \mathsf{any}\}$$

- peer: an object that is referenced by a peer reference x is part of the same representation as the current object. In other words, the two objects have the same owner.
- rep: an object that is referenced by a rep reference x is part of the current (i.e., this) object's representation. In other words, the current object is the *owner* of the object referenced by x.
- any: the any qualifier does not provide any information about the ownership of the object.

In addition, there are two other qualifiers used internally by the type system: lost and self. lost expresses that the result of viewpoint adaptation cannot be expressed statically. self is the only type for the implicit parameter this.

Detailed explanations and examples of these qualifiers can be found in [9]. The qualifiers form the following subtyping hierarchy:

$$\mathsf{rep} <: \mathsf{lost} \qquad \mathsf{self} <: \mathsf{peer} <: \mathsf{lost} \qquad \mathsf{lost} <: \mathsf{any}$$

Viewpoint Adaptation Function. Viewpoint adaptation in Universe Types is as follows:

$$\begin{array}{llll}
\mathsf{self} & \rhd \_ & = \_ & \\
\_ & \rhd \mathsf{self} & = \_ & \\
\mathsf{peer} & \rhd \mathsf{peer} & = \mathsf{peer} & \\
\mathsf{rep} & \rhd \mathsf{peer} & = \mathsf{rep} & \\
\_ & \rhd \mathsf{any} & = \mathsf{any} & \\
q & \rhd q' & = \mathsf{lost} & \text{otherwise}
\end{array}$$

The context of adaptation is always *the type of the receiver* at both field access and method call. For example, the context of adaptation at y.f is the type of y; similarly, the context at call y.m(z) is receiver the type of receiver y as well. Adapting to the context of the receiver can be interpreted as object sensitivity.

Additional Constraints Universe Types impose additional constraints. In our framework, these constraints are expressed by parameters $\mathcal{B}$:

$$\begin{array}{lll}
\mathcal{B}_{(\text{TNEW})}(q_l, q_r) & = & \{q_r \neq \mathsf{any}\} \\
\mathcal{B}_{(\text{TWRITE})}(q_r, q_f, q_o) & = & \{q_r \neq \mathsf{any}, q_r \rhd q_f \neq \mathsf{lost}\} \\
\mathcal{B}_{(\text{TCALL})}(\mathsf{m}, q_r, q_o) & = & \text{let } \mathit{typeof}(\mathsf{m}) = q \to q' \text{ in} \\
 & & \quad \text{if } \mathit{impure}(\mathsf{m}) \text{ then} \\
 & & \quad\quad \{q_r \neq \mathsf{any}, q_r \rhd q \neq \mathsf{lost}\} \\
 & & \quad \text{else} \\
 & & \quad\quad \{q_r \rhd q \neq \mathsf{lost}\}
\end{array}$$

impure is a predicate that returns true if method m is pure, and false otherwise. $\mathcal{B}$ for all other rules are empty.

ReIm and Universe Types are two systems that can be instantiated in the framework. Others are Ownership Types [2], Non-null Types [6], AJ [14] and EnerJ [12]. Once a type system is specified in the framework, the types can be inferred by the unified type inference algorithm (Section 2.3).

2.3 Unified Type Inference

Type inference for pluggable type systems is difficult because these systems allow many different valid typings. For example, in ReIm, one can type every variable mutable and the program will be well-typed. However, this "easy" typing is not desirable because, intuitively, we would like to prove a maximal number of variables as being readonly, not mutable. The main challenges for type inference lie in (1) understanding the most desirable, i.e., *"best" typing*, and (2) computing the "best" typing.

Surprisingly, a simple idea works. We first compute a *set-based solution* S which maps variables to *sets* of type qualifiers. The set-based solver initializes every unannotated variable to the maximal set of qualifiers, and every annotated variable to the singleton set that contains the programmer-provided qualifier. There is a transfer function f_s for each statement s. Each f_s takes as input the current mapping S and outputs an updated mapping S'. f_s refines the set of each reference that participates in s as follows. Let $\mathsf{x}, \mathsf{y}, \mathsf{z}$ be the references in s. For each reference, say x, f_s removes each $q_\mathsf{x} \in S(\mathsf{x})$ from $S(\mathsf{x})$, if there does not exist a pair $q_\mathsf{y} \in S(\mathsf{y})$, $q_\mathsf{z} \in S(\mathsf{z})$ such that $q_\mathsf{x}, q_\mathsf{y}, q_\mathsf{z}$ type check under the type rule for s from Figure 1. For example, consider statement x = y.f and corresponding rule (TREAD). Suppose that $S(\mathsf{x}) = \{\mathsf{polyread}\}$, $S(\mathsf{y}) = \{\mathsf{readonly}, \mathsf{polyread}, \mathsf{mutable}\}$ and $S(\mathsf{f}) = \{\mathsf{readonly}, \mathsf{polyread}\}$ before the application of the transfer function. The transfer function removes readonly from $S(\mathsf{y})$ because there does not exist $q_\mathsf{f} \in S(\mathsf{f})$ that satisfies $\mathsf{readonly} \rhd q_\mathsf{f} <: \mathsf{polyread}$. Similarly, it removes readonly from $S(\mathsf{f})$ because $\_ \rhd \mathsf{readonly} = \mathsf{readonly}$, is not a subtype of polyread as rule (TREAD) requires. After the application of the transfer function, S' is as follows: $S'(\mathsf{x}) = \{\mathsf{polyread}\}$, $S'(\mathsf{y}) = \{\mathsf{polyread}, \mathsf{mutable}\}$, and $S'(\mathsf{f}) = \{\mathsf{polyread}\}$.

The resulting set-based solution contains all valid typings in the program. The question is, how do we extract the "best" typing out of it? Again, there is a simple idea that works. The programmer specifies a *preference ranking* over the type qualifiers. For example, for ReIm the ranking is

$$\mathsf{readonly} > \mathsf{polyread} > \mathsf{mutable}$$

Informally, this means that, if possible, we type a variable readonly, If not possible, we type it polyread. Finally if neither readonly nor polyread are possible, we type it mutable.

More formally, the preference ranking over qualifiers establishes a ranking over all valid typings. A typing T is better than a typing T' if and only if T has more readonly variables than T', or T and T' have the same number of readonly variables but T has more polyread variables than T'. Thus, the typing with the most readonly variables is the "best", and the typing with all mutable variables is the worst. The *maximal typing* is the typing extracted from the set-based solution by mapping each variable to the maximally preferred qualifier in its set. If the maximal typing type checks, then this is the unique "best" typing. For ReIm, the maximal typing provably always type checks.

The above approach generalizes to many context-sensitive pluggable type systems. For several interesting systems such as Universe Types [5] and AJ [14] the maximal typing provably type checks and we easily get the "best" typing. For other systems, such as Classical Ownership Types [2], the maximal typing does not always type check. Still, the approach is useful because it reduces the annotation burden on programmers — we have been able to type large programs using about 6 manual annotations per KLOC on average.

Computing the "best" typing is a constraint problem. Dietl et al. [4] present type inference for Generic Universe Types (GUT), where the GUT constraints are encoded as a boolean satisfiability problem and solved with a Max-SAT solver. This approach can infer "best" typing for any pluggable type system. The key difference between the Max-SAT-based approach and our inference is that the former is exponential, while the latter is low polynomial (assuming a small number of qualifiers, the complexity of inference is $O(n^2)$).

In previous work [10], we instantiated ReIm in the framework and evaluated it on 13 Java benchmarks, comprising 766,053 lines of code in total. The inference took less than 1 minute in our largest benchmark xalan, which has more than 300KLOC. We also instantiated UT in the framework in [9] and it showed almost identical performance results as ReIm. All experiments were performed on a server with Intel® Xeon® CPU X3460 @2.80GHz and 8 GB RAM (the maximal heap size is set to 2 GB).

3. RELATED WORK

Our most closely related paper is [9]. The paper describes instantiation of the framework for two well-known context-sensitive pluggable type systems, Universe Types [5] and Ownership Types [2]. Our goal is to generalize the framework beyond ownership-like type systems. We envision that programmers can not only infer types and make use of existing type systems, but also easily prototype new type systems.

The most closely related work outside of our work is JQual [8] by Greenfieldboyce and Foster. JQual supports source-sink systems, where there are two qualifiers (the source and the sink) related through subtyping. The programmer annotates variables with the source or the sink qualifier, and JQual infers the rest of the qualifiers. Our framework differs in several ways. First, it is more general. It allows for more complex, context-sensitive (i.e., polymorphic) type systems such as ownership. The framework expresses context-sensitivity by using viewpoint adaptation and polymorphic qualifiers (polyread in ReIm and peer in UT). Second, it scales better. This is because it encodes context sensitivity directly in the type system, using viewpoint adaptation; this allows for better flexibility (essentially, one can tailor the context sensitivity to the needs of the specific system). Previous work [9, 10] has shown promising scalability of the framework. JQual uses a "behind-the-scene" flow analysis to infer qualifiers. Although JQual's flow analysis has a field-sensitive/context-sensitive mode, it scales only in its field-insensitive/context-insensitive mode according to [1].

4. EXPECTATIONS

Our expectations from the FSE community are twofold. Firstly, we would like to learn about new interesting flow properties and design type systems that specify and infer these properties. Confidentiality is one such property. Programmers annotate some variables as confidential and other variables as unsafe; the system infers the rest of the types and proves or disproves the desired confidentiality.

Secondly, we would like to advertise the framework and gather feedback on usability. Existing type systems (e.g., ownership types, immutability types, non-null types, and many others) have widely-acknowledged and well-known software engineering benefits. Newer systems such as AJ and EnerJ address important problems in concurrency and energy efficient computing. However, to the best of our knowledge, these systems are not used in software engineering practice. We conjecture that the lack of practical adoption is due, to some extent, to the annotation burden. Our framework significantly alleviates, and in many cases completely removes the annotation burden on programmers.

The current implementation is publicly available at website `http://code.google.com/p/type-inference/`.

5. REFERENCES

[1] S. Artzi, A. Kieżun, J. Quinonez, and M. D. Ernst. Parameter reference immutability: formal definition, inference tool, and comparison. *Automated Software Engineering*, 16(1):145–192, Dec. 2009.

[2] D. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64, 1998.

[3] W. Dietl, S. Dietzel, M. D. Ernst, K. Muslu, and T. Schiller. Building and Using Pluggable Type-Checkers. In *ICSE*, 2011.

[4] W. Dietl, M. D. Ernst, and P. Müller. Tunable static inference for generic universe types. In *ECOOP*, pages 333–357, 2011.

[5] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.

[6] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA*, pages 302–312, 2003.

[7] C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, and J. Duffy. Uniqueness and reference immutability for safe parallelism. In *OOPSLA*, 2012.

[8] D. Greenfieldboyce and J. S. Foster. Type qualifier inference for java. In *OOPSLA*, pages 321–336, 2007.

[9] W. Huang, W. Dietl, A. Milanova, and M. D. Ernst. Inference and checking of object ownership. In *ECOOP*, pages 181–206, 2012.

[10] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst. ReIm & ReImInfer: Checking and inference of reference immutability and method purity. In *OOPSLA*, 2012.

[11] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, Jan. 2005.

[12] A. Sampson, W. Dietl, and E. Fortuna. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI*, pages 164–174, 2011.

[13] M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In *OOPSLA*, pages 211–230, 2005.

[14] M. Vaziri, F. Tip, J. Dolby, C. Hammer, and J. Vitek. A type system for data-centric synchronization. In *ECOOP*, pages 304–328, 2010.