Report

Tunable universe type inference

Author(s):
Dietl, Werner; Ernst, Michael; Müller, Peter

Publication Date:
2009

Permanent Link:

https://doi.org/10.3929/ethz-a-006843105 →

Tunable Universe Type Inference

Technical Report 659

Department of Computer Science, ETH Zurich

Werner Dietl, Michael Ernst, Peter Müller

University of Washington
{wmdietl,mernst}@cs.washington.edu

ETH Zurich
Peter.Mueller@inf.ethz.ch

December 2009

Tunable Universe Type Inference

Werner Dietl[1], Michael Ernst[1], and Peter Müller[2]

[1] University of Washington
{wmdietl,mernst}@cs.washington.edu
[2] ETH Zurich
Peter.Mueller@inf.ethz.ch

Abstract. Object ownership is useful for many applications such as program verification, thread synchronization, and memory management. However, even lightweight ownership type systems impose considerable annotation overhead, which hampers their widespread application. This paper address this issue by presenting a tunable static type inference for Universe types. In contrast to classical type systems, ownership types have no single most general typing. Therefore, our inference is tunable: users can indicate a preference for certain typings by configuring heuristics through weights. A particularly effective way of tuning the static inference is to obtain these weights automatically through runtime ownership inference. We present how the constraints of the Universe type system can be encoded as a boolean satisfiability (SAT) problem, how the runtime ownership inference produces weights from program executions, and how a weighted Max-SAT solver finds a correct Universe typing that optimizes the weights. Our implementation provides the static and runtime inference engines, as well as a visualization tool.

1 Introduction

Heap structures are hard to understand and reason about. Aliasing—multiple references to the same object—makes errors all too common. For example, it permits the mutation of an object through one reference to be observed through other references. Aliasing makes it hard to build complex object structures correctly and to guarantee invariants about their behavior. This leads to problems in many areas of software engineering, including modular verification, concurrent programming, and memory management.

Object ownership [13] structures the heap hierarchically to control aliasing and access between objects. Ownership type systems aid in the understanding of heap structures and enforce proper encapsulation of objects. However, ownership type systems require considerable annotation overhead, which can be a significant burden for software engineers.

Helping software engineers to transition from un-annotated programs to code that uses an ownership type system is crucial to facilitate the widespread application of ownership type systems. Standard techniques for static type inference [16] are not applicable. First, there is no need to check for the existence of a correct typing; such a typing trivially exists by having a flat ownership structure.

Second, there is no notion of a best or most general ownership typing. In realistic implementations, there are many possible typings and corresponding ownership structures, and the preferred one depends on the intent of the programmer. Ownership inference needs to support the developer in finding desirable structures by suggesting possible structures and allowing the programmer to guide the inference.

This paper presents a static inference for the Universe type system [15]. The static inference builds a constraint system that is solved by a SAT solver. An important virtue of our approach is that the static inference is *tunable*; the SAT solver can be provided with weights that express the preference for certain solutions. These weights can be determined by general heuristics (for instance, to prefer deep ownership for fields and general typings for method signatures), by partial annotations, or through interaction with the programmer.

Runtime ownership inference is a particularly effective way to determine weights automatically. The runtime inference executes the program, for example using the available tests, and observes the generated object structures. It then uses a dominator algorithm to determine the deepest possible ownership structure and to find possible Universe annotations. The quality of the annotations determined by the runtime inference depends on the code coverage, which can be reflected in the weights for the suggested annotations. By combining the runtime inference with the static inference, we get the best of both approaches: the static inference ensures that the inferred typing is correct, while the runtime inference obtains weights for the static inference that ensure a deep ownership structure, which is more likely to reflect design intent and to be useful to programmers.

The main contributions of this paper are:

Static Inference: an encoding of the Universe type rules into a constraint system that can be solved efficiently by a SAT solver to find possible annotations.

Tunable Inference: combining the static inference with a weighted Max-SAT solver and using the runtime inference, heuristics, and programmer interaction to determine weights.

Runtime Inference: using information from actual executions to determine the deepest possible ownership structures and suggest annotations to the static inference.

Prototype: an implementation of our inference scheme as a set of command-line tools and Eclipse plug-ins that allow simple interaction with and visualization of the inferred typing.

The outline of this paper is as follows. Sec. 2 overviews the architecture of the system. Sec. 3 presents the static inference, including support for partial annotations and heuristics. It also motivates why the static inference alone is not enough; Sec. 4 follows up with the runtime inference. Sec. 5 describes our prototype implementation and our experience with it. Finally, Sec. 6 discusses related work, and Sec. 7 concludes.

```
public class Person {
    peer Person spouse;
    rep  Account savings;

    int assets() {
        any Account a = spouse.savings;
        return savings.balance + a.balance;
    }
}
```

Fig. 1: A simple example with Universe types.

2 Overview

This section overviews Universe types and our tunable static inference.

2.1 Universe Type System

For simplicity, this paper uses the non-generic Universe type system [18, 15]. Generic Universe types lead to a more complex constraint system that can be handled by the same inference approach.

Statically, the Universe type system (UTS) associates with each reference type one of three *ownership modifiers* to describe the ownership structure. The modifier `peer` expresses that the current object `this` is in the same context as the referenced object, the modifier `rep` expresses that the current object is the owner of the referenced object, and the modifier `any` does not give any static information about the relationship of the two objects. A reference with an `any` modifier conveys less information than the same references with a `peer` or `rep` modifier; therefore, an `any`-modified type is a supertype of the `peer` and `rep` versions. Fig. 1 illustrates the use of these modifiers. A `Person` object owns its savings account and has the same owner as its spouse.

An important concept in the Universe type system is *viewpoint adaptation*. An ownership modifier expresses ownership relative to the current object `this`. When the interpretation of the current object changes, for example when we access a field through a reference other than `this`, we need to adapt the ownership modifier to this new viewpoint. We define viewpoint adaptation as a function $\triangleright$ that takes two ownership modifiers and yields the adapted modifier. This paper only discusses a simplified version and considers three cases: (1) `peer` $\triangleright$ `peer` = `peer`; (2) `rep` $\triangleright$ `peer` = `rep`; and (3) for all other combinations the result is `any`. For instance, the ownership modifier of `spouse.savings` in Fig. 1 is determined by viewpoint adaptation of the modifier of `spouse` (`peer`) and the modifier of `savings` (`rep`), which yields `any`. We refer to previous papers [18, 15, 17] for a detailed description.

The Universe type system enforces the *owner-as-modifier* discipline: An object o may be referenced by any other object, but reference chains that do not pass through o's owner must not be used to modify o. This allows owner objects

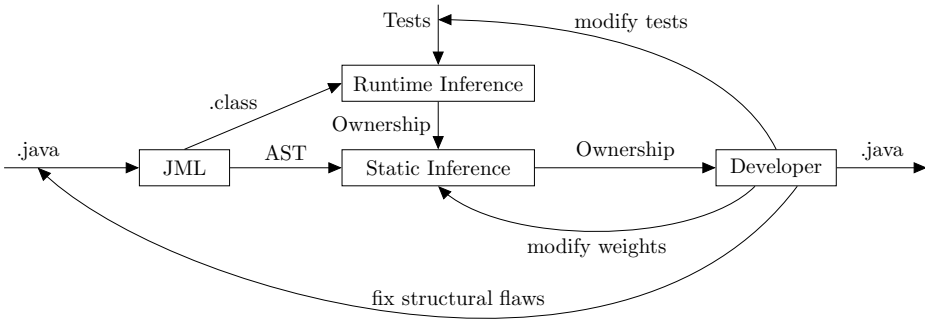


Fig. 2: Architecture of our tunable inference approach.

to control state changes of owned objects and thus maintain invariants. The owner-as-modifier discipline is enforced by forbidding field updates and non-pure (side-effecting) method calls through `any` references. An `any` reference can still be used for field accesses and to call pure (side-effect-free) methods. For instance, method `assets` in Fig. 1 may read the `balance` field via the `any` reference `a`, but would not be allowed to update it.

The Universe type system minimizes annotation overhead by using the default modifier `peer` for most references. This default makes the conversion from Java to Universe types simple, since all programs continue to compile. However, it results in a flat ownership structure. Inference is still required, in order to find a deep ownership structure.

2.2 Architecture

Fig. 2 illustrates the architecture of our tunable inference approach.

Our tools take, as input, the purity of methods. We re-implemented [24] Sălcianu's algorithm [45] for this task.

We use the Java Modeling Language (JML) [28] compiler to read source code, build an abstract syntax tree (AST), and generate the Java byte code needed for the runtime inference. The JML compiler supports the Universe type system and handles source code that is partially annotated with ownership modifiers.

The static inference performs syntax-directed constraint generation. For each possible occurrence of an ownership modifier in the source code, it creates a constraint variable. For example, there is a constraint variable for the ownership modifier in each field declaration, and one for the ownership modifier in each `new` expression. For each AST node, it creates a constraint over these variables, which correspond one-to-one to the type rules of the Universe type system [18, 15, 17].

The runtime inference is an optional step that tunes the static inference. It traces program executions and uses the object graphs deduced from these executions to determine ownership modifiers. The runtime inference determines the deepest possible ownership hierarchy for a particular program run. As with

any dynamic analysis, the inference result is dependent on the coverage of the test suite. For instance, if a class does not get instantiated during the program run, the runtime inference cannot determine ownership modifiers for its fields. Therefore, our system treats the results of the runtime inference as suggestions that are encoded in weights that influence how the static inference chooses among multiple valid options for ownership annotations.

The static inference encodes the constraints and the weights into the input format of a SAT solver, runs the SAT solver, decodes the reply back into a constraint variable assignment, and presents it to the programmer. The developer has four options: (1) accept the annotations and let the tool insert them into the source code; (2) modify the weights of the static inference to encourage or force certain results; (3) add or modify test cases to improve the results of runtime inference; (4) fix defects in the source code that the analysis reveals.

3 Tunable Static Inference

The static inference has two independent parts: building the constraint system from the abstract syntax tree (Sec. 3.2) and then encoding the constraints into the input format of the SAT solver (Sec. 3.3).

3.1 Programming Language

We use a simple, Java-like programming language to present the static inference. Fig. 3 summarizes the syntax of the language and the naming conventions. A sequence of A elements is denoted as $\overline{A}$.

A program P consists of a sequence of class declarations $\overline{Cls}$, the name of a main class C, and a main expression e. A program execution instantiates an instance of class C and executes expression e with this instance as the current object. A class declaration *Cls* consists of the name of the class and superclass and of field and method declarations. Field declarations are simple pairs of types and identifiers. Method declarations consist of the method purity, the return type, the method name, the parameter declaration, and an expression for the method body. An expression e can be the `null` literal, a method parameter access, object creation, field read, field update, method call, or cast.

A type T is a pair consisting of an ownership modifier u and a class name C. The definition of the ownership modifiers is the only deviation from previous formalizations of the UTS [15, 17]. In addition to the three ownership modifiers `peer`, `rep`, and `any`, we add ownership constraint variables α. These ownership variables are used as placeholders for the concrete modifiers that we want to infer.

3.2 Building the Constraints

The constraints are built in a syntax-directed manner, by traversing the AST and creating the ownership variables α and constraints over them. Our algorithm creates a variables for each location where an ownership modifier may

$$
\begin{array}{rcl}
P & ::= & \overline{Cls},\ C,\ e \\
Cls & ::= & \texttt{class}\ Cid\ \texttt{extends}\ C\ \{\ \overline{fd}\ \overline{md}\ \} \\
C & ::= & Cid \mid \texttt{Object} \\
fd & ::= & T\ f; \\
md & ::= & p\ T\ m(\overline{mpd})\ \{\ e\ \} \\
p & ::= & \texttt{pure} \mid \texttt{impure} \\
mpd & ::= & T\ pid \\
e & ::= & \texttt{null} \mid x \mid \texttt{new}\ T() \mid e.f \mid e_0.f{:=}e_1 \mid e_0.m(\overline{e}) \mid (T)\ e \\
T & ::= & u\ C \\
u & ::= & \boxed{\alpha} \mid \texttt{peer} \mid \texttt{rep} \mid \texttt{any} \\
x & ::= & pid \mid \texttt{this}
\end{array}
$$

pid	parameter identifier
f	field identifier
m	method identifier
Cid	class identifier
$\boxed{\alpha}$	ownership variable identifier

Fig. 3: Syntax of our programming language. We use constraint variables α (framed) as placeholders for ownership modifiers. The definition of the ownership modifiers is the only deviation from previous formalizations of the UTS.

occur. Then, it creates constraints that correspond to the type rules expressed abstractly over the variables. Our inference is a type-based analysis [41] that runs only on valid Java programs. Therefore, we do not encode all Java type rules, but can restrict the constraints to the additional checks that are necessary for the Universe type system.

3.2.1 The kinds of constraint.

Five kinds of constraint are necessary:

Declaration ($decl(\alpha)$): we generate a declaration constraint for every ownership variable α. This ensures that all internal constraints for the encoding are enforced.

Subtype ($\alpha_1 <: \alpha_2$): enforces that variable α_1 will be assigned an ownership modifier that is a subtype of the ownership modifier assigned to α_2. `peer` and `rep` are subtypes of `any` and are unrelated to one another. This constraint is needed for assignments and pseudo-assignments such as parameter passing and return statements.

Adaptation ($\alpha_1 \triangleright \alpha_2 = \alpha_3$): is a relation between three variables and ensures that the viewpoint adaptation of variable α_2 from the viewpoint expressed by α_1 results in α_3.

Equality & Inequality ($\alpha_1 = \alpha_2$, $\alpha_1 \neq \alpha_2$): in certain situations we need to forbid or fix an ownership modifier, for example, a non-pure method call is only allowed on `peer` or `rep` receivers, so we forbid `any` for the variable representing the receiver.

Comparable ($\alpha_1 <:> \alpha_2$): if the program contains reference comparisons or casts, these also create a relationship between the two variables. It is not allowed that one variable is assigned `peer` and the other `rep`, because such variables are never comparable.

3.2.2 Rules for constraint generation. Fig. 4 defines the creation of the constraints for a program. It defines judgments over class, field, and method declarations and over expressions. These judgments determine a set of constraints Σ that have to hold for the program.

An environment Γ maps variables to their types. Function *env* defines this mapping depending on the surrounding class and the method parameter declarations.

We now discuss the rules of Fig. 4 in turn.

A program declaration determines all constraints for the class declarations and for the main expression. The environment Γ maps `this` to C.

The constraints for a class declaration consist of the constraints for the field and method declarations.

For field and method parameter declarations, the rules add a declaration constraint for the variable.

A method declaration ensures that the ownership variables appearing in the return type and method parameter types are declared, that the constraints imposed by the method body are enforced, and that the type of the method body is a subtype of the return type. Function $\mathit{overriding}(\mathit{Cid}, m)$ ensures that, if the current method is overriding a method in a superclass, the parameter and return types are consistent.

Finally, there are 7 judgments for expressions.

The `null` literal, and accessing a method parameter or `this`, do not impose constraints.

For a cast, the constraints of the expression must hold, and the two types must be are castable. Note that the rule only infers annotations for existing casts and does not introduce additional casts.

An object creation expression ensures that the ownership variable is declared and that an ownership modifier different from `any` is used.

The field access judgment ensures the constraints from the receiver expression and the viewpoint adaptation. Below, we define the helper functions *fType*, to look up the field type, and *mType*, for method signature after viewpoint adaptation.

A field update additionally ensures that the right-hand side is a subtype of the left-hand side and that the receiver expression is not assigned `any`.

The rule for a method call expression ensure the constraints from the subexpressions and viewpoint adaptation, and that the type of the argument expression is a subtype of the parameter type. If the method is non-pure, it additionally ensures that the receiver is not `any`.

We now define the helper functions.

Program declaration: $\boxed{\vdash P : \Sigma}$

$$\frac{\vdash \overline{Cls} : \Sigma_c \qquad \Gamma \vdash e : \Sigma_e}{\vdash \overline{Cls}, C, e : \_, (\Sigma_c, \Sigma_e)}$$

Class declaration: $\boxed{\vdash Cls : \Sigma}$

$$\frac{\vdash \overline{fd} : \Sigma_f \qquad Cid \vdash \overline{md} : \Sigma_m}{\vdash \texttt{class}\ Cid\ \texttt{extends}\ C\ \{\ \overline{fd}\ \overline{md}\ \} : (\Sigma_m, \Sigma_f)}$$

Field and Method Parameter Declaration: $\boxed{\vdash T\ f : \Sigma}$, $\boxed{\vdash T\ pid : \Sigma}$

$$\frac{}{\vdash u\ C\ \_ : decl(u)}$$

Method Declaration: $\boxed{Cid \vdash md : \Sigma}$

$$\frac{\begin{array}{cc} \vdash \overline{mpd} : \Sigma_p & env(Cid, \overline{mpd}) = \Gamma \\ \Gamma \vdash e : T, \Sigma_b & overriding(Cid, m) = \Sigma_o \\ \Sigma = \Sigma_b, \Sigma_p, \Sigma_o, om(T) <: om(T_r), decl(om(T_r)) & \end{array}}{Cid \vdash p\ T_r\ m(\overline{mpd})\ \{\ e\ \} : \Sigma}$$

Expression: $\boxed{\Gamma \vdash e : T, \Sigma}$

$$\frac{}{\Gamma \vdash \texttt{null} : T, \emptyset} \qquad \frac{}{\Gamma \vdash x : \Gamma(x), \emptyset} \qquad \frac{\Gamma \vdash e_0 : u_0\ C_0, \Sigma \qquad \Sigma' = \Sigma, decl(u), u <:> u_0}{\Gamma \vdash (u\ C)\ e_0 : u\ C, \Sigma'}$$

$$\frac{\Sigma = decl(u), u \neq \texttt{any}}{\Gamma \vdash \texttt{new}\ u\ C() : u\ C, \Sigma} \qquad \frac{\Gamma \vdash e : T_0, \Sigma_0 \qquad fType(T_0, f) = T, \Sigma_1}{\Gamma \vdash e.f : T, (\Sigma_0, \Sigma_1)}$$

$$\frac{\begin{array}{c} \Gamma \vdash e_0 : T_0, \Sigma_0 \\ \Gamma \vdash e_1 : T_1, \Sigma_1 \\ fType(T_0, f) = T_2, \Sigma_2 \\ \Sigma = om(T_1) <: om(T_2), om(T_0) \neq \texttt{any} \end{array}}{\Gamma \vdash e_0.f{:=}e_1 : T_2, (\Sigma_0, \Sigma_1, \Sigma_2, \Sigma)} \qquad \frac{\begin{array}{c} \Gamma \vdash e_0 : T_0, \Sigma_0 \\ \Gamma \vdash e_1 : T_1, \Sigma_1 \\ mType(T_0, m) = p\ T_r\ m(T_p\ pid), \Sigma_2 \\ \Sigma = om(T_1) <: om(T_p) \\ p = \texttt{impure} \Rightarrow \Sigma' = \Sigma, om(T_0) \neq \texttt{any} \\ p = \texttt{pure} \Rightarrow \Sigma' = \Sigma \end{array}}{\Gamma \vdash e_0.m(e_1) : T_r, (\Sigma_0, \Sigma_1, \Sigma_2, \Sigma')}$$

Helper:

$$om(u\ C) = u \qquad\qquad env(\overline{T\ pid}) = \overline{pid \mapsto T}$$

Fig. 4: Constraint generation rules.

```
class C {
       α1 Object f = new α2 Object();
}
```

Fig. 5: Example with two variables α_1 and α_2 for which we want to determine ownership modifiers.

Function $fType(C, f)$ returns the declared field type of field f in class C or a superclass of C. It only returns a type and does not introduce an additional constraint. Function $fType(u\ C, f)$ determines the type of field f adapted from viewpoint $u\ C$ to `this`. It results in an adapted field type and a constraint between the viewpoint variable and the variable representing the declared type.

$$\begin{aligned} &fType(u\ C, f) = u'\ C', (u \rhd u'' = u', decl(u')) \\ &\quad \text{where } fType(C, f) = u''\ C' \end{aligned}$$

Function $mType(C, m)$ returns the declared method signature of method m in class C or a superclass of C. It, again, only returns a method signature and does not need to add a constraint. Function $mType(u\ C, m)$ determines the method signature of method m adapted from viewpoint $u\ C$ to `this`. It results in an adapted method signature and a constraint between the viewpoint variable and the variables representing the declared type parameter and return type, respectively.

$$\begin{aligned} &mType(u\ C, m) = p\ u'_r\ C_r\ m(u'_p\ C_p\ pid), \Sigma \\ &\quad \text{where } \begin{array}{l} mType(C, m) = p\ u_r\ C_r\ m(u_p\ C_p\ pid) \\ \Sigma = u \rhd u_r = u'_r, u \rhd u_p = u'_p, decl(u'_r), decl(u'_p) \end{array} \end{aligned}$$

3.2.3 Example. The example of Fig. 5 illustrates the constraint generation process. This class contains a field declaration and a field initializer consisting of an object creation. This simplicity allows us to illustrate every step.

We want to infer two variables: the modifier for the field declaration α_1 and the modifier for the object creation α_2. The rules create the following constraints:

1. $decl(\alpha_1)$: variable α_1 is a legal modifier for an instance field declaration,
2. $decl(\alpha_2), \alpha_2 \neq$ `any`: variable α_2 is a legal modifier for an object creation, and
3. $\alpha_2 <: \alpha_1$: variable α_2 is a subtype of variable α_1.

3.3 Encoding for a SAT Solver

Once we have the constraint system Σ, it needs to be solved. We decided to use an existing weighted Max-SAT solver for three reasons. First, the Universe type system allows only three ownership modifiers; thus, constraints can easily be encoded as boolean formulas. Second, the weights allow us to encode heuristics that direct the SAT solver to produce "good" solutions. Third, reusing a solver allows us to benefit from all the optimizations that went into existing solvers.

Constraint	CNF Encoding
$decl(\alpha)$	$(\neg\beta^{peer} \vee \neg\beta^{rep}) \wedge (\neg\beta^{peer} \vee \neg\beta^{any}) \wedge (\neg\beta^{rep} \vee \neg\beta^{peer}) \wedge$ $(\neg\beta^{rep} \vee \neg\beta^{any}) \wedge (\neg\beta^{any} \vee \neg\beta^{rep}) \wedge (\neg\beta^{any} \vee \neg\beta^{peer}) \wedge$ $(\beta^{peer} \vee \beta^{rep} \vee \beta^{any})$
$\alpha_1 \rhd \alpha_2 = \alpha_3$	$(\neg\beta_1^{peer} \vee \neg\beta_2^{peer} \vee \beta_3^{peer}) \wedge (\neg\beta_1^{rep} \vee \neg\beta_2^{peer} \vee \beta_3^{rep}) \wedge$ $(\neg\beta_1^{any} \vee \beta_3^{any}) \wedge (\neg\beta_2^{any} \vee \beta_3^{any}) \wedge (\neg\beta_2^{rep} \vee \beta_3^{any})$
$\alpha_1 <: \alpha_2$	$(\neg\beta_1^{any} \vee \beta_2^{any}) \wedge (\neg\beta_2^{peer} \vee \beta_1^{peer}) \wedge (\neg\beta_2^{rep} \vee \beta_1^{rep})$
$\alpha_1 <:> \alpha_2$	$(\neg\beta_1^{peer} \vee \neg\beta_2^{rep}) \wedge (\neg\beta_1^{rep} \vee \neg\beta_2^{peer})$
$\alpha = u$	β^u
$\alpha \neq u$	$\neg\beta^u$

Fig. 6: For each kind of constraint (see Sec. 3.2.1), the CNF formula that encodes it.

This section explains how to encode the constraints Σ as conjunctive normal form (CNF) formulas, which is the input format of the SAT solver. Our implementation supports changing the solver or the encoding of the constraints, which facilitates experimentation. In the following, we explain one possibility for encoding the constraints in CNF.

The SAT solver either returns an assignment of booleans that satisfies the formula or notifies the user that the formula is un-satisfiable. From the assignment of booleans we can determine ownership modifiers for the variables that satisfy all constraints.

The encoding of the constraints is defined in Fig. 6 and is explained in the following.

3.3.1 Declaration. The CNF formula uses three booleans β^{peer}, β^{rep}, and β^{any} to represent a variable α from the constraints. Two booleans would be sufficient to encode the three possibilities. However, the constraints are simpler, and therefore more efficiently solvable, when using a one-hot encoding of the options [21]. In Fig. 6, the first two lines ensure that only at most one of these three booleans is assigned true; it is the CNF form of the following:

$$\begin{array}{l}(\beta^{peer} \Rightarrow (\neg\beta^{rep} \wedge \neg\beta^{any})) \wedge \\ (\beta^{rep} \Rightarrow (\neg\beta^{peer} \wedge \neg\beta^{any})) \wedge \\ (\beta^{any} \Rightarrow (\neg\beta^{rep} \wedge \neg\beta^{peer}))\end{array}$$

The third line ensures that at least one of the variables is true.

3.3.2 Viewpoint Adaptation. The first line contains the two clauses that ensure that concrete ownership information is preserved. The three clauses in

the second line ensure that the result of the viewpoint adaptation is `any` in the other cases. These conditions are the CNF form of the following implications:

$$(\beta_1^{peer} \wedge \beta_2^{peer} \Rightarrow \beta_3^{peer}) \wedge$$
$$(\beta_1^{rep} \wedge \beta_2^{peer} \Rightarrow \beta_3^{rep}) \wedge$$
$$(\beta_1^{any} \Rightarrow \beta_3^{any}) \wedge (\beta_2^{any} \Rightarrow \beta_3^{any}) \wedge (\beta_2^{rep} \Rightarrow \beta_3^{any})$$

3.3.3 Subtype. If the subtype is `any`, then the supertype is `any`. If the supertype is `peer`, then the subtype is `peer`. If the supertype is `rep`, then the subtype is `rep`. This can be expressed as:

$$(\beta_1^{any} \Rightarrow \beta_2^{any}) \wedge (\beta_2^{peer} \Rightarrow \beta_1^{peer}) \wedge (\beta_2^{rep} \Rightarrow \beta_1^{rep})$$

which can be reformulated in CNF to give the formula in Fig. 6.

3.3.4 Comparable. The clauses forbid that one variable is assigned `peer` when the other variable is assigned `rep`.

3.3.5 Equality & Inequality. These constraints are simply encoded by forcing that the corresponding boolean is either true or false.

Once all constraints are encoded as CNF formula, we can pass it to a SAT solver, which will return a legal assignment, if the constraint system is solvable. The booleans of this assignment can then be interpreted as the assignments to the ownership variables α.

3.4 Heuristic Choice of a Solution

In our simple example from Fig. 5, we create a constraint system consisting of 6 boolean variables and 18 clauses (7 clauses for the field type, 8 clauses for the object creation, 3 clauses for the subtype relation). Legal assignments for this constraint system are:

α_1	α_2
`peer`	`peer`
`rep`	`rep`
`any`	`peer`
`any`	`rep`

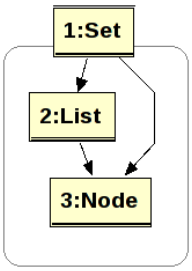
The SAT solver may return any of these, depending on its search strategy, initialization of the boolean variables, ordering of variables and clauses, etc.

For any program, a possible solution is to assign `peer` to all variables. This is not useful (unless it is the only possibility), because it corresponds to a completely flat structure.

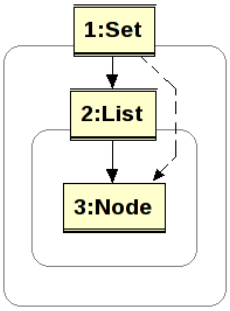
A human programmer is influenced by a variety of design considerations when choosing the ownership modifiers. A deeper ownership structure gives better

```
public class Set {
       Node lastUsed;
       List impl;
}
public class List {
       Node first;
}
public class Node {
       Node next;
}
```

(a) Code



(b) Flat



(c) Deep

Fig. 7: A class `Set` using a `List`, which uses `Node`s. The `lastUsed` field implements a caching optimization that speeds up repeated membership queries. In parts (b) and (c), contexts are depicted by rounded rectangles, and owner objects sit atop the context of objects they own.

encapsulation, but limits sharing. The types in method signatures influence what other objects can call the method. We want the SAT solver to give us a solution that respects these design considerations.

Our approach is to use the weight feature of weighted Max-SAT solvers. These solvers permit a user to express a preference for each boolean to be true or false, as a weight. A maximal-weight assignment that satisfies all constraints is not only correct, but is useful and is close to what a human developer would want (assuming the weights are set appropriately).

Our tool combines weights from multiple heuristics, by scaling and then adding their weights. The programmer can influence the weights and scaling of these different heuristics.

One simple, and frequently effective, heuristic is to prefer deeper ownership structures over flatter structures. One can try to express this by assigning a weight of, say, 50 to the `rep` booleans, a weight of 10 to `peer` booleans, and 0 to the `any` booleans. Then a solution with the maximum weight is supposed to represent the deepest structure that is possible. However, the heuristic alone does not guarantee the deepest structure overall. The source code in Fig. 7a illustrates this problem. It implements a set using a list, which in turn uses nodes.

Using the above weights, the SAT solver might assign `rep` to both fields in class `Set` and `peer` to the fields in `List` and `Node`, giving a weight of 120 and the structure in Fig. 7b. However, what a programmer would prefer is most likely the structure in Fig. 7c, which assigns `rep` to field `impl` and `any` to field `lastUsed` of class `Set`, `rep` to field `first` of class `List`, and `peer` to field `next` of class `Node`, but only has a weight of 110. This illustrates that our simple heuristic does not give the desired structure.

In the next section, we will explain how a runtime inference can be used to determine the deepest possible ownership structure. While a static heuristic applies equally to all elements of a certain category (for instance, all fields), the

runtime inference obtains weights for individual occurrences, such as a particular field.

4 Runtime Inference

The inference of Universe types from program executions is performed in the following five steps:

1. Build the representation of the object store
2. Build the dominator tree
3. Resolve conflicts with the Universe type system
4. Harmonize different instantiations of a class
5. Output Universe types

We use the classes in Fig. 8 to illustrate how the algorithm works. This is an artificial example to illustrate all aspects of the algorithm.

```
public class Demo {
       public static void main(String[] args) {
              new Demo().testA(args.length > 0);
       }
       public void testA(boolean b) { new A(b); }
}
class A { B b;
       boolean mod;

       A(boolean m) { mod = m; b = new B(this); }
       void off() { mod = false; }
}
class B { C c;
       Object o = new Object();

       B(A a) { c = new C(a); }
}
class C { A a;

       C(A na) { a = na; if (a.mod) { a.off(); } }
}
```

Fig. 8: Running example to illustrate the runtime inference algorithm.

The main class is `Demo`; the Java entry-point `main` creates an instance of class `Demo` and calls method `testA` on that instance. The argument is a boolean that depends on the number of command line arguments. Method `testA` creates an `A` instance. Class `A` stores the boolean flag and creates an instance of class `B`.

Class `B` creates a `C` instance and a `java.lang.Object` instance. Finally, class `C` stores a reference to the `A` object it receives and depending on the value of the `mod` field calls the `off` method on the `A` instance.

4.1 Build the Representation of the Object Store

From a program execution we get a sequence of modifications of the object store. Instead of looking at only single snapshots of the store, we build a cumulative representation of the object store. This *Extended Object Graph* (EOG) [46] represents all objects that ever existed in the store, all references between these objects that were ever observed, and, in particular, which objects modified which other objects. The information about modifications is particularly important since Universe types do not restrict references in general (unlike other ownership type systems), but the modification of objects.

For each object in the EOG, we record information about its fields as well as the parameters and results of its methods. We use this information to infer ownership modifiers for these variables.

We distinguish between two types of references in the EOG: write references and naming references. *Write references* are used to update a field or call a non-pure method on an object; these references mainly determine the ownership structure of an application. In addition we store references that were only used for reading fields and calling pure methods. These *naming references* are needed to map the resulting EOG back to the source code.

For example, a call x`.foo(`y`)` introduces two edges in the EOG. A write references from the current receiver object `this` to x represents that `this` modifies x by calling the non-pure method `foo`. This reference will later influence the ownership relation between `this` and x. A naming reference from x to y represents that a method of x takes y as parameter. This naming reference is labeled with the name of the formal parameter and will later be used to infer the ownership modifier of the parameter.

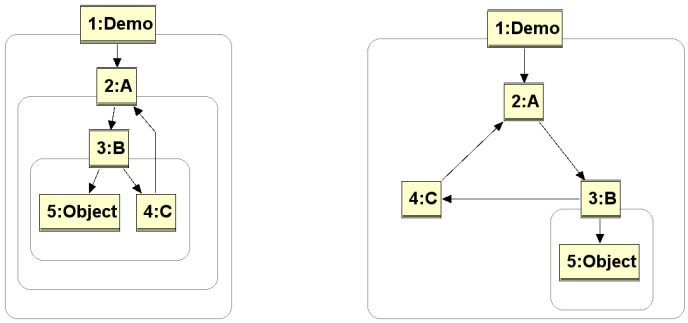
In our running example (Fig. 8), class `A` contains in its constructor the statement `b = new B(this)`. On the bytecode level, this corresponds to two steps, first the creation of a new object and then the update of the field `b` of the current object. For an object creation, we insert a write edge from the current receiver object to the newly created object. This write edge ensures that the ownership modifier for the object creation is either `peer` or `rep`, a requirement of the Universe type system. For a field update, we store a write reference from the current object to the receiver of the field update and a naming reference from the receiver of the field update to the object on the right-hand side. The naming reference is labeled with the field name. All naming references for a field can later be used to infer the ownership modifier for that field.

4.2 Build the Dominator Tree

Universe types require that all modifications of an object are initiated by its owner. For the EOG, this means that all chains of write references from the

root object to an object x must go through x's owner. Therefore, we can identify suitable candidates for the owner of x by computing the dominators of x. The concept of dominators is well-known in the compiler field [4], and efficient algorithms have been developed [30].

Universe types do not restrict references that are merely used for reading. Therefore, the naming references in the EOG do not carry information that helps us to determine ownership relations between objects. Consequently, we ignore them when we build the dominator graph.



(a) Dominator Tree (b) After conflict resolution

Fig. 9: Object graphs for the source code from Fig. 8.

The result of finding the dominators for the example from Fig. 8 is shown in Fig. 9a. Domination is depicted by rounded rectangles. A direct dominator sits atop the rounded rectangle that groups the objects it dominates. It is a candidate for becoming the owner of this group of objects.

4.3 Resolve Conflicts with the Universe Type System

Domination is a good approximation of ownership, but it cannot be directly used to infer Universe types. The Universe type system only allows write references within a context and from an owner to an owned object. On the other hand, a dominator graph can have references from an object to an object in an enclosing context. Such write references are not permitted in the Universe type system. If such references are found in the EOG, the involved objects are raised to a common level until no more conflicts are present.

This situation is illustrated by the code in Fig. 8. If we observe an execution of the constructor of class `C` when `a.mod` is `false` then the `off` method is not called on the `a` reference. In this case, the reference from object 4 to object 2 is used in a read-only manner, that is, the EOG contains a naming reference between object 4 and object 2. Under this assumption, the dominator graph in Fig. 9a is a valid ownership structure. The reference between object 4 and

object 2 is stored in field `a` of class `C`. This field will be annotated with an `any` ownership modifier.

However, if `a.mod` is `true`, the non-pure method `off` is called on `a`. This results in a write reference from object 4 to object 2. In this case, the dominator graph does not represent a valid ownership structure because there is a write reference to an object in an enclosing context. This write reference can neither be typed with a `rep` nor with a `peer` modifier and is, therefore, not admissible in Universe types. To solve this problem, we flatten the ownership structure to make the write reference from object 4 to object 2 admissible. This is done by raising the origin of the write reference (object 4) to the context that contains the destination of the write reference (object 2). This makes the two objects peers, and the write reference between them is admissible since it can be typed with modifier `peer`.

However, raising object 4 creates a conflict for the write reference from object 3 to object 4 since now object 4 is neither owned by nor a peer of object 3. Therefore, we apply the same solution again; this time, object 3 is raised to be in the same context as object 4. The resulting dominator graph is depicted in Fig. 9b. In this graph, all write references are from a direct dominator to an object it dominates or between objects with the same direct dominator. Therefore, this graph represents a valid ownership structure that can be expressed in Universe types.

Our example shows that conflict resolution has to be applied repeatedly because resolving one conflict can cause others. Nevertheless, conflict resolution can be implemented efficiently without visiting the same write reference twice. To achieve this, we use a list of conflicting write references and process the list in a top-down way, that is, objects higher-up in the dominator graph are processed first. Moreover, we resolve conflicts that cross a large number of context boundaries before conflicts that cross fewer contexts.

4.4 Harmonize Different Instantiations of a Class

After conflict resolution, the EOG is consistent with the owner-as-modifier discipline. However, it might not be possible to statically type the EOG because different instances of a class might be in different ownership relations. To enforce uniformity of all instances of a class, we traverse all instances of each class and compare the ownership properties of each variable (field or parameter). This step has to take into account both write and naming references in the EOG.

If for any given variable the ownership relations are the same (for instance, they all point to peer objects), the variable can be typed statically. If they differ, we apply a resolution that is similar to the conflict resolution described in the previous subsection. If at least one instance of a variable is the origin of a peer reference and the other instances of this variable are rep references, we raise the targets of the rep references to make them peers and type the variable with modifier `peer`. If at least one instance of a variable is the origin of a reference that is neither a peer nor a rep reference, the variable is typed with modifier

`any`. In this case, downcasts are needed at the point where this variable is used for field updates and calls to non-pure methods.

For example, imagine that method `testA` in class `Demo` is once called with `false` and once with `true` as the argument. Then we have two instances of class `A`, once with a deep ownership structure as in Fig. 9a and once with a flat structure as in Fig. 9b. The annotation for field `b` in class `A` is once `rep` and once `peer`. The algorithm then decides to use `peer` as annotation for field `b` and raises the non-conforming instance to a higher level. Because we raise an object together with all peers that reference it or are referenced by it, this step cannot create new conflicts in the ownership graph.

4.5 Output Universe Types

After the first four steps of the runtime inference algorithm, we have determined possible ownership modifiers for field declarations, method parameters and results, and allocation expressions. For the example in Fig. 7a, the runtime inference determines the deep ownership structure depicted in Fig. 7c, provided that the `Set` object 1 does not directly modify object 3.

Local variables are not inferred from the EOG because that would require monitoring every assignment of a local variable. However, this gap is closed by the subsequent static inference. The static inference also detects if the modifiers determined by the runtime inference violate the type rules of Universe types. This problem occurs in particular when the runtime inference is based on program runs with insufficient code coverage. The static inference then uses weights to determine which results of the runtime inference it should override. To facilitate this step, our runtime inference not only compute suggestions for ownership modifiers but also provides a weight for each suggested modifier that indicates the confidence in the correctness of this particular suggestion, based on the code coverage.

5 Discussion & Implementation

In this section, we discuss some technical details and outline aspects of our implementation.

5.1 Discussion

Both the static and the runtime inference can operate on small program fragments, for instance, a class and its unit tests. However, to infer meaningful ownership modifiers, it is often necessary to consider larger contexts, in particular, the clients of a class. Nevertheless, the heuristics of the static inference encourage solutions that are usable in a wider range, for example, by preferring method parameters to have the `any` modifier.

As we explained above, the ownership assignment obtained from the runtime inference might not be correct, but the combination with static inference removes

this problem. For the static inference, we encode all type rules in SAT, so if the solution is satisfiable, the annotated program will compile. An incorrect input from the runtime inference might then only cause a longer exploration of the state space.

Note, however, that Universe types support casts. Downcasts that specialize ownership information (that is, casts from `any` to `peer` or `rep`) require a runtime check. Our static inference does not guarantee these runtime checks succeed. To mitigate the risk of a runtime error, we allow the static inference to determine ownership modifiers for the cast expressions present in the input program, but not to introduce additional casts. Information from the runtime inference might allow one to further reduce the risk of runtime errors.

Both the static and the runtime inference also support arrays and static methods. Arrays in the Universe type system use two ownership modifiers, one for the relation between `this` and the array object, and one for the relation between `this` and the objects stored in the array. Static method calls take an ownership modifier that determines the relationship between the current object and the execution of the static method.

5.2 Static Inference

The architecture of the static inference tool is separated into three components: (1) the AST visitor, (2) the constraint builder, and (3) the constraint encoder.

The AST visitor encapsulates all interaction with the AST and calls the constraint builder functions corresponding to the different AST elements. It uses the JML2 compiler [28], which supports Universe type annotations in its AST. The input source is parsed with support for ownership modifiers, but the type rules are not checked. This allows us to parse partially-annotated programs that might not conform to the type rules.

The constraint builder creates the constraints for the different syntax elements. It encapsulates the type rules for the Universe type system from the AST and the solver that is used. It also collects the weighting information from the different heuristics and the runtime inference.

The constraint encoder takes a constraint system and encodes it for a particular constraint solver. We implemented an encoder that creates CNF formulas in the DIMACS format and the weighting information in the .cnf.pb format supported by the PBS tool [7]. The constraint encoder also decodes the replies from the solver and maps the solutions back to ownership modifiers.

5.3 Runtime Inference

The runtime inference is split into two parts: a tracing agent that monitors the execution of Java programs and the inference tool which determines the ownership modifiers from (multiple) trace files.

A Java Virtual Machine Tooling Interface (JVMTI) agent written in C monitors the Java Virtual Machine (JVM) execution of the program. The agent

receives events from the virtual machine and produces a trace file that documents the execution of the program. The trace file is in a simple XML format. Storing the execution of a program in a trace file gives the following advantages: (1) Multiple trace files can be generated to achieve good code coverage. (2) Interactive or long-running programs need to be traced only once for each desired code path. This trace file can then be reused later without requiring human interaction or recomputing results. On the other hand, storing the trace files on disc and then parsing them again in the next phase could lead to a performance overhead; we leave optimizations of this aspect as future work.

The main inference tool is an independent Java application that performs the steps described in Sec. 4. It reads (multiple) trace files generated by the tracing agent and builds one Extended Object Graph from the available information. Then the dominators are determined, conflicts are resolved, multiple instances are harmonized, and the output is written to an XML file or passed to the static inference. The different steps of the algorithm are implemented as visitors that manipulate the EOG, allowing us to simply change the inference. For more details see our earlier work [19, 32].

5.4 Annotation Management

All tools can be used on the command line and are configured using XML configuration files. The output of the inference tools is an annotation XML file that contains the ownership modifiers for the encountered types. This separate annotation file makes the comparison of results of multiple runs simple. If the source code of the traced program is available then the annotations can be inserted into the source code using a separate annotation tool we developed. Producing the output in XML also allows us to support several annotation formats, for instance, the existing JML2 Universe syntax and the JSR 308-style Java annotations [20].

5.5 Eclipse Integration

The goal of our inference approach is to allow the programmer to easily tune the inference results. The command-line tools are good for batch-processing and evaluations, but not suitable for developers. To ease the usage of the inference tools we created a set of Eclipse 3.5 plug-ins for the Java development environment. Fig. 10 shows a screenshot.

The different inference properties can be easily configured from the project settings and multiple inference settings can be stored.

The execution of the runtime inference is integrated into the normal "Run as" approach. A graphical visualizer using the Eclipse Graphical Editing Framework (GEF) displays the extended object graph while it is built up and modified by the runtime inference. This gives a clear understanding of how the program executes and how the runtime inference algorithm works. Note that the graphs in Figs. 7 and 9 are screenshots from this tool.

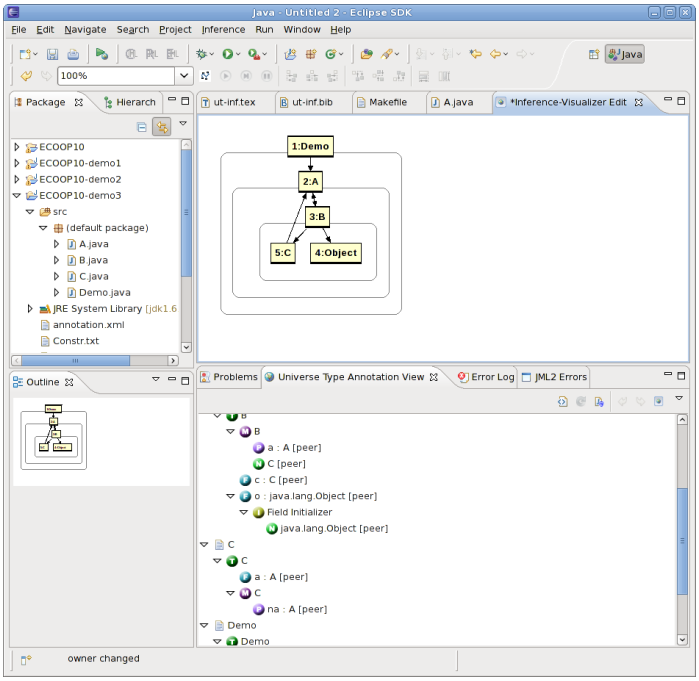


Fig. 10: A screenshot showing a runtime object graph on top and the static inference on the bottom.

The static inference allows the simple selection of heuristics and the programmer can easily fix ownership modifiers for certain variables and re-run the inference, even without parsing the whole source code again.

The annotation tools allow the programmer to easily review the inferred ownership modifiers and insert them into the source code.

Finally, the JML2 compiler is also integrated into Eclipse and the Universe type checker can be run from within Eclipse.

6 Related Work

SafeJava [8, 10, 9] provides intra-procedural type inference for local variables to reduce the annotation overhead. Agarwal and Stoller [3] describe a run-time technique that infers further annotations. AliasJava [6] uses a constraint system to infer alias annotations. Another static analysis for ownership types resulted in a large number of ownership parameters [27]. Kacheck/J [26] infers package-level encapsulation properties.

Alisdair Wren's work on inferring ownership [46] provided a theoretical basis for our work on runtime inference. It developed the idea of the Extended

Object Graph and how to use the dominator as a first approximation of ownership. It builds on ownership types [12, 5, 8, 13] which uses parametric ownership and enforces the owner-as-dominator discipline. The number of ownership parameters for parametric type systems is not fixed and is usually determined by the programmer, as is the number of type parameters for a class. Trying to automatically infer a good number of ownership parameters makes their system complex.

Milanova [35] presents preliminary results for the static inference of Universe types. Her tool applies a static alias analysis to construct a static object graph and then computes dominators to obtain candidates for owners. This approach is similar to our earlier work on runtime inference [19].

Abi-Antoun and Aldrich [2, 1] present how runtime object graphs can be extracted from programs with ownership domain annotations to visualize the architecture of the program. Noble [40] focuses on the treatment of aliasing in heap visualizations.

The box model [42] separates the program into module interfaces and implementations. Ownership annotations are still required for the module interface, but are automatically inferred for the implementations. It might be possible to adapt our runtime inference to help with the annotation of the interfaces.

Pedigree types [31] present an intricate ownership type system similar to Universe types with polymorphic type inference for annotations. It builds a constraint system that is reduced to a set of linear equations. The inference does not help with finding good ownership structures, but only helps propagate existing annotations. We believe that our approach to type inference is easier to understand and better supports the programmer in finding the desired ownership structure.

Rayside et al. [44] present a dynamic analysis that infers ownership and sharing, but they do not map the results back to an ownership type system. Mitchell [36] analyzes the runtime structure of Java programs and characterizes them by their ownership patterns. The tool handles heaps with 29 million objects and creates succinct graphs. The tool Yeti [37] analyzes heap snapshots and helps in understanding large heaps and finding memory leaks. Both tools do not distinguish between read and write references and the results are not mapped to an ownership type system.

Daikon [22] is a tool to detect likely program invariants from program traces. Invariants are only enforced at the beginning and end of methods and therefore also snapshots are only taken at these spots. From these snapshots we cannot infer which references were used for reading and which were used for writing. Therefore we could not directly use Daikon, but our runtime inference tool has a similar architecture and we are looking into possible synergies.

The work on uniqueness and ownership inference [33] presents a static analysis to infer these program properties, without mapping the results to a type system. General type qualifier inference [25] presents relevant work for qualifier inference; however, applied to ownership, it would not help in the inference of

the deepest or most desirable ownership structure, but infers any solution that satisfies all constraints, possibly a flat structure.

The system most similar to ours is a type inference systems against races [23]. It builds a constraint system and uses a SAT solver to find solutions and use a Max-SAT encoding to produce good error reports, in cases where the constraint system is unsatisfiable. However, they are not concerned with finding an optimal structure for their system, since any valid locking strategy is acceptable. We use the weighting mechanism to find a desirable ownership structure also for satisfiable solutions.

SAT solvers have seen a wide variety of uses for other optimization problems over the last several years [34].

Ownership has been used to verify object invariants in Spec# [29] and JML [38]. These verification systems encourage, but do not enforce the use of ownership to encapsulate the state an invariant depends on. Therefore, we could use the invariants as another source of suggestions for ownership modifiers.

7 Conclusions

We presented a novel approach to static ownership inference that uses a Max-SAT solver to optimize the result w.r.t. preferences encoded as weights. The use of weights allows us to take into account tentative ownership information from various sources such as heuristics, partial annotations, and in particular runtime inference. Our initial experiments suggest that the combination of static and runtime inference leads to a practical approach, which produces correct typings with deep ownership structures.

Now that we have a working inference tool, our highest priority for future work is performing a larger case study to evaluate the quality of the inferred typings and to investigate what ownership structures occur in real programs.

Universe types were an ideal target system for our work because its modifiers can easily be encoded in boolean formulas. As future work, we plan to apply our approach to other ownership systems to explore four avenues. First, we plan to extend our approach to Generic Universe Types (GUT) [17]. The key issues are to adapt the runtime inference to GUT, especially in the presence of an erasure semantics, and to explore whether we can infer ownership topologies without assuming an encapsulation discipline. Second, we plan to extend our inference to support ownership transfer [14, 39], which requires static inference to infer uniqueness of variables and runtime inference to cope with dynamic changes of ownership information. Third, we plan to investigate how our approach can be adapted to ownership-parametric type systems [6, 13, 43]. We are confident that by combining static and runtime inference, we can effectively determine the minimum number of ownership parameters required to type a class. Fourth, we plan to explore how we can infer ownership annotations for more complex topologies such as ownership domains [5] or multiple ownership [11].

References

1. M. Abi-Antoun and J. Aldrich. Compile-time views of execution structure based on ownership. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2007.
2. M. Abi-Antoun and J. Aldrich. Static extraction of sound hierarchical runtime object graphs. In *Types in Language Design and Implementation (TLDI)*, 2009.
3. R. Agarwal and S. D. Stoller. Type Inference for Parameterized Race-Free Java. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 2937 of *LNCS*, pages 149–160. Springer-Verlag, 2004.
4. A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools, Second Edition*. Addison-Wesley, 2007.
5. J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 1–25. Springer-Verlag, 2004.
6. J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 311–330. ACM Press, 2002.
7. F. Aloul. PBS v2.1: Incremental Pseudo-Boolean Backtrack Search SAT Solver and Optimizer, 2003. `http://www.aloul.net/Tools/pbs/`.
8. C. Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, MIT, 2004.
9. C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 211–230. ACM Press, 2002.
10. C. Boyapati, A. Salcianu, W. Beebee Jr., and M. Rinard. Ownership types for safe region-based memory management in real-time Java. In *Programming Language Design and Implementation (PLDI)*, pages 324–337. ACM Press, 2003.
11. N. Cameron, S. Drossopoulou, J. Noble, and M. Smith. Multiple ownership. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 441–460. ACM Press, 2007.
12. D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 292–310. ACM Press, 2002.
13. D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM Press, 1998.
14. D. Clarke and T. Wrigstad. External uniqueness is unique enough. In L. Cardelli, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 2743 of *LNCS*. Springer-Verlag, 2003.
15. D. Cunningham, W. Dietl, S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. Universe types for topology and encapsulation. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects (FMCO)*, volume 5382 of *LNCS*, pages 72–112. Springer-Verlag, 2008.
16. L. Damas and R. Milner. Principal type-schemes for functional programs. In *Principles of programming languages (POPL)*, pages 207–212. ACM Press, 1982.
17. W. Dietl. *Universe Types: Topology, Encapsulation, Genericity, and Tools*. PhD thesis, Department of Computer Science, ETH Zurich, 2009.
18. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, 2005.

19. W. Dietl and P. Müller. Runtime Universe Type Inference. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2007.
20. M. D. Ernst. Type annotations specification (JSR 308). `http://pag.csail.mit.edu/jsr308/`, September 12, 2008.
21. M. D. Ernst, T. D. Millstein, and D. S. Weld. Automatic SAT-compilation of planning problems. In *Joint Conference on Artificial Intelligence (IJCAI)*, pages 1169–1176, Nagoya, Aichi, Japan, August 23–29, 1997.
22. M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007.
23. C. Flanagan and S. N. Freund. Type inference against races. In *SAS*, pages 116–132. Springer-Verlag, 2004.
24. D. Graf. Implementing purity and side effect analysis for Java programs. Semester Project, Department of Computer Science, ETH Zurich, Winter 2005/06.
25. D. Greenfieldboyce and J. S. Foster. Type qualifier inference for Java. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 321–336. ACM Press, 2007.
26. C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. In *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 241–253, 2001.
27. S. E. Moelius III and A. L. Souter. An object ownership inference algorithm and its application. In Marco T. Morazan, editor, *Mid-Atlantic Student Workshop on Programming Languages and Systems*, 2004.
28. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, D. M. Zimmerman, and W. Dietl. JML reference manual. `www.jmlspecs.org`, 2008.
29. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 491–516. Springer-Verlag, 2004.
30. T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141, 1979. Available from `http://doi.acm.org/10.1145/357062.357071`.
31. Y. D. Liu and S. Smith. Pedigree types. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2008.
32. F. Lyner. Runtime Universe type inference. Master's thesis, Department of Computer Science, ETH Zurich, 2005.
33. K. Ma and J. S. Foster. Inferring aliasing and encapsulation properties for Java. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 423–440. ACM Press, 2007.
34. J. Marques-Silva. Practical applications of boolean satisfiability. In *Workshop on Discrete Event Systems (WODES)*. IEEE Press, May 2008.
35. A. Milanova. Static inference of Universe Types. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2008.
36. N. Mitchell. The runtime structure of object ownership. In D. Thomas, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 4067 of *LNCS*, pages 74–98. Springer-Verlag, 2006.

37. N. Mitchell, E. Schonberg, and G. Sevitsky. Making sense of large heaps. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 5653 of *LNCS*, pages 77–97. Springer-Verlag, 2009.
38. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.
39. P. Müller and A. Rudich. Ownership transfer in Universe Types. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 461–478. ACM Press, 2007.
40. J. Noble. Visualising objects: Abstraction, encapsulation, aliasing, and ownership. In *Software Visualization*, volume 2269 of *LNCS*, pages 607–610. Springer-Verlag, 2002.
41. J. Palsberg. Type-based analysis and applications. In *Program Analysis for Software Tools and Engineering (PASTE)*, pages 20–27, 2001.
42. A. Poetzsch-Heffter, K. Geilmann, and J. Schäfer. Infering ownership types for encapsulated object-oriented program components. In *Program Analysis and Compilation, Theory and Practice*, volume 4444 of *LNCS*, pages 120–144. Springer-Verlag, 2007.
43. A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic ownership for generic Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 311–324. ACM Press, October 2006.
44. D. Rayside, L. Mendel, and D. Jackson. A dynamic analysis for revealing object ownership and sharing. In *Workshop on Dynamic Analysis (WODA)*, pages 57–64. ACM Press, 2006.
45. A. Salcianu and M. C. Rinard. Purity and side effect analysis for Java programs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3385 of *LNCS*, pages 199–215. Springer-Verlag, 2005.
46. A. Wren. Inferring ownership. Master's thesis, Department of Computing, Imperial College, June 2003. `http://www.cl.cam.ac.uk/~aw345/`.