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

First Version: December 2009
Second Version: March 2010

Tunable Universe Type Inference

Werner Dietl
University of Washington
wmdietl@cs.washington.edu

Michael D. Ernst
University of Washington
mernst@cs.washington.edu

Peter Müller
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 addresses 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 tools as a set of command-line tools and Eclipse plug-ins, which we used to experiment with the proposed workflow and the quality of our inference.

Keywords Ownership, Universe types, inference, static, runtime

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, aliasing 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 program verification, concurrent programming, and memory management.

Object ownership [12] structures the heap hierarchically to control aliasing and access between objects. Ownership type systems express properties of the heap topology, for instance whether two instances of a list may share node objects. Such information is needed to show the correctness of a coarse-grained locking strategy, where the lock of the list protects the state of all its nodes [7]. Ownership type systems also enforce encapsulation, for instance, by forcing all modifications of an object to be initiated by its owner. Such guarantees are useful to maintain invariants that relate the state of multiple objects [38]. However, ownership type systems require considerable annotation overhead, which is 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 [15] 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 [14], a lightweight ownership type system designed to enable program verification [37]. Our 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 solution is type correct; 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 to suggest annotations to the static inference.

Prototype: an implementation of our inference scheme as a set of command-line tools and Eclipse plug-ins, which we used to experiment with the proposed workflow and the quality of our inference.

This paper is organized as follows. Sec. 2 gives background on the Universe type system. Sec. 3 overviews the architecture of the system. Sec. 4 presents the static inference, including support for partial annotations and heuristics. It also motivates why the static inference alone is not enough; Sec. 5 follows up with the runtime inference. Sec. 6 describes our prototype implementation and our experience with it. Finally, Sec. 7 discusses related work, and Sec. 8 concludes.

2. Background on Universe Types

The Universe type system (UTS) [17, 14] is a simple ownership type system that is integrated into the tool suite of the Java Modeling Language (JML) [28]. It organizes the heap hierarchically into contexts and restricts modifications across context boundaries. In the UTS, like in most other ownership systems, each object has at most one owner object. A *context* is the set of objects sharing an owner. The ownership relation is acyclic.

The UTS expresses the ownership topology by associating one of three *ownership modifiers* with each reference type in a program. 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.

An ownership modifier expresses ownership relative to the current receiver object `this`. When the interpretation of the current object changes, for example when a field is accessed through a reference other than `this`, we need to adapt the ownership modifier to this new viewpoint. For instance, the `rep` modifier in the declaration of field `savings`

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

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

Figure 1: A simple example with Universe types. A `Person` object owns its savings account and has the same owner as its spouse.

in Fig. 1 indicates that the `Account` object is owned by the `Person` object that contains the field. Therefore, it would be wrong to assign the modifier `rep` to the field access `spouse.savings`, because that would express that the account is owned by `this`, which is in general different from the object `spouse` containing the field. The UTS determines the type of a field access by adapting the modifier of the field to the new viewpoint. In our example, this viewpoint adaptation from `spouse` to `this` yields the modifier `any`, which reflects correctly that the `Account` object is neither owned by `this` nor a peer of `this`. We define this *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 (an extension to the more sophisticated viewpoint adaptation used in the formalization of the UTS [16, 14] is straightforward): (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`.

The Universe type system enforces that programs adhere to the heap topology described by the ownership modifiers. In addition, the UTS enforces an encapsulation scheme called *owner-as-modifier* discipline [17]: 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 to control state changes of owned objects and thus maintain invariants. For instance, a `Person` object can easily maintain the invariant `savings.balance >= 0` because the owner-as-modifier discipline guarantees that aliases to the `Account` object `savings` cannot be used to modify the `Account` object, in particular, its `balance` field. Therefore, it is sufficient to check that each method of the `Person` object maintains the invariant.

The owner-as-modifier discipline is enforced by forbidding field updates and non-pure (side-effecting) method calls through an `any` reference. For instance, the call `spouse.savings.withdraw(1000000)` is rejected by the type

system because the viewpoint-adapted modifier of the receiver, `spouse.savings`, is `any`. 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`.

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 required, to find a deep ownership structure automatically.

3. Overview

The overall workflow of our inference tool is illustrated in Fig. 2. The four main components are the static inference, the runtime inference, the purity inference, and the user interface. The purity inference provides purity information for methods, which is required to enforce the owner-as-modifier discipline. It is a re-implementation [23] of Sălcianu and Rinard's algorithm [47]. This section gives an overview of the other three components and describes how the programmer interacts with the tool.

Static inference. The static inference performs syntax-directed constraint generation. The main input is an abstract syntax tree (AST) of the Java program, created by the JML compiler. It supports Universe types and, therefore, handles Java programs with partial Universe annotations.

For each possible occurrence of an ownership modifier in the source code, the inference 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 [17, 14, 16].

The constraints are solved using a weighted Max-SAT solver. The weights are obtained from general heuristics and, if available, prior runs of the runtime inference. The output of the inference is a complete annotation of the input program, that is, an ownership modifier for each reference type in the program. The annotations can be persisted in the Java source code for use by downstream tools such as a program verifier.

Runtime inference. The runtime inference is an optional step that tunes the static inference. It takes as input the executable program, a set of test cases, and purity information. The inference tool then executes the test cases, traces the 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 runtime inference emits the ownership modifiers it determines together with information on the code coverage. The static inference uses this information as *suggestions* whose weights depend on the test coverage of the runtime inference. The static inference tool has the freedom to override the suggestions from runtime inference, for instance, if those suggestions would not lead to a valid UTS program.

User interface. As we have explained in the introduction, there is generally no best typing in ownership type systems. Therefore, we expect the inference tool to be used iteratively. The user interface (an Eclipse plug-in) allows users to control the inference process by guiding the static inference (through partial annotations and the choice of heuristics) and by selecting test cases and object graph files for the runtime inference. Moreover, the user interface includes a basic visualizer for runtime and static object graphs and their ownership structure. Programmers can use this visualizer to inspect the result of the inference.

If the annotations do not fully reflect the programmers design intent, the programmer can optimize the inference process in four ways. First, the programmer can add ownership modifiers to the input program to express their design intent explicitly. For instance, the programmer might decide to declare a `Person`'s `spouse` field with a `peer` modifier even if the constraints of the type system would permit the undesirable situation where a person owns their spouse. Our inference tools treat partial annotations as constraints that have to be satisfied. Second, the programmer might decide to modify the weights of the static inference to encourage or force certain results. For instance, the programmer might select heuristics that favor general types for a library in order to make the library as widely applicable as possible, whereas they might select heuristics that favor restrictive modifiers when the whole program is available to facilitate subsequent use of the ownership information, for instance, by a program verifier. Third, the programmer might decide to add or modify test cases to improve the results of runtime inference. This action seems useful when the inferred types are generally not satisfying. By storing object graphs persistently, our tool enables programmers to run additional tests and to combine their results with the object graphs from previous runs. Fourth, the programmer might decide to fix defects in the source code. For instance, Universe types encourage a layered design and prevent lower layers from calling non-pure methods of higher layers. The absence of a satisfying ownership annotation might indicate violations of such an architecture.

Once the programmer is satisfied with the obtained results, our tool inserts the ownership modifiers into the source code to improve the documentation of the code, to encourage that the heap topology and encapsulation are considered

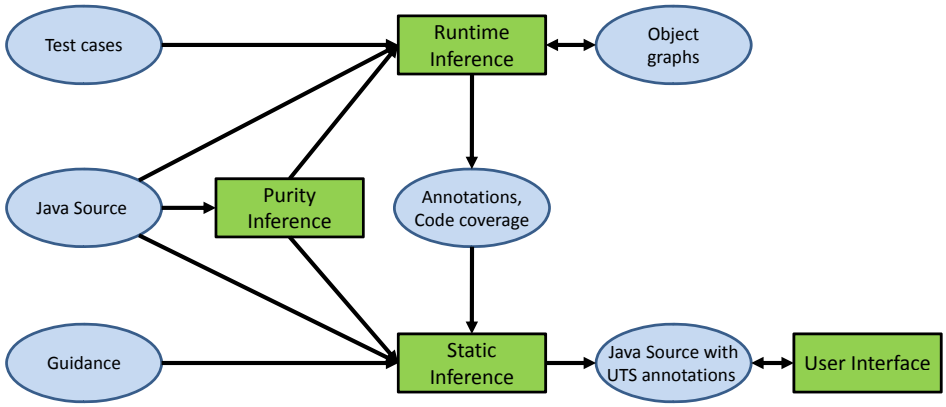


Figure 2: Architecture and workflow of our tunable inference. Components and files are depicted by boxes and ovals, respectively.

during program maintenance, and to make them available to downstream tools.

4. Tunable Static Inference

Universe types express ownership information via three ownership modifiers. This lightweight approach makes it possible to infer ownership information statically by encoding the constraints of the type system in a boolean formula that can then be solved by a SAT solver. Using a SAT solver is not only a very efficient way to solve the constraints but also allows us to use weights to direct the solver towards preferable solutions. This section introduces a simple Java-like programming language, explains how our system extracts constraints from a program, and shows how the constraints are encoded into the input format of the SAT solver.

4.1 Programming Language

Fig. 3 summarizes the syntax of the language and the naming conventions.

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 [14, 16]. In addition to the three ownership modifiers `peer`, `rep`, and `any`, we add ownership constraint variables

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

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

Figure 3: Syntax of our programming language. Constraint variables α (framed) are placeholders for ownership modifiers. The definition of the ownership modifiers is the only deviation from previous formalizations of the UTS. A sequence of A elements is denoted as $\overline{A}$.

α. These ownership variables are used as placeholders for the concrete modifiers that the system will infer.

4.2 Building the Constraints

The constraints are built in a syntax-directed manner. For each reference type occurring in the program (that is, for each position where an ownership modifier may occur), we introduce an *ownership variable* α that represents the ownership modifier of that reference type. Our inference will later

assign one of the three ownership modifiers `rep`, `peer`, or `any` to each of these ownership variables.

The system creates constraints on these ownership variables, which correspond to the type rules expressed abstractly over the ownership variables. Our inference is a type-based analysis [42] that runs only on valid Java programs. Therefore, this paper does not encode all Java type rules, but gives only constraints for the additional checks for the Universe type system.

The kinds of constraint. The inference rules (which will be introduced later, and which appear in Fig. 4) make use of five kinds of constraints.

Declaration ($decl(\alpha)$): Declaration constraints indicate the introduction of a fresh ownership variable α. The system generates one declaration constraint for each occurrence of a reference type in the program.

Subtype ($\alpha_1 <: \alpha_2$): A subtype constraint 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. Subtype constraints are used for assignments, parameter passing, result passing.

Adaptation ($\alpha_1 \rhd \alpha_2 = \alpha_3$): An adaptation constraint ensures that the viewpoint adaptation of variable α_2 from the viewpoint expressed by α_1 results in α_3. Adaptation constraints are used wherever viewpoint adaptation occurs in the type system, that is, for field accesses, parameter passing, and result passing.

Equality & Inequality ($\alpha_1 = u$, $\alpha_1 \neq u$): An equality constraint fixes the value of an ownership variable α_1 to the ownership modifier u. They are used to handle input programs with partial annotations.

An inequality constraint forbids certain values for an ownership variable. They occur whenever the type system disallows certain modifiers, for instance, the `any` modifier for the receiver of field updates.

Comparable ($\alpha_1 <:> \alpha_2$): A comparable constraint expresses that two ownership modifiers are not incompatible, that is, one could be a subtype of the other. These constraints are used to prevent casts that are known to fail at runtime. In UTS, the only two modifiers that are not comparable are `rep` and `peer`.

Rules for constraint generation. Fig. 4 contains the rules for extracting constraints from a program. It defines judgments over class, field, and method declarations, as well as 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. Function om gives the ownership modifier for a type. We use _ to denote elided elements.

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. In this rule, _ stands for a field identifier f or a parameter identifier pid. A method declaration requires 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 $overriding(Cid, m)$ requires that, if the current method is overriding a method in a superclass, the parameter and return types are consistent.

Finally, there are seven judgments for expressions. The `null` literal, and accessing a method parameter or `this`, do not impose constraints. We discuss casts immediately below. An object creation expression requires that the ownership variable is declared and that the ownership modifier is different from `any`. The field access judgment requires the constraints from the receiver expression and the viewpoint adaptation. The adaptation constraint is determined by the helper function $fType$, which yields the field type after viewpoint adaptation, see below. A field update additionally requires that the right-hand side to be a subtype of the left-hand side and the receiver expression to be different from `any`. The rule for a method call expression requires the constraints from the subexpressions and viewpoint adaptation (analogous to $fType$, the helper function $mType$ yields the method signature after viewpoint adaptation together with the necessary constraints, see below). Moreover, method calls require that the type of the argument expression is a subtype of the parameter type and, if the method is non-pure, that the modifier of the receiver is different from `any`.

For a cast, the constraints of the subexpression must hold, and the two types must be comparable. Note that our inference determines ownership modifiers for casts that already exist in the program, but does not introduce any additional casts. For instance, if the inferred modifier of variables `x` and `o` are `peer` and `any`, respectively, then the constraint for the expression `x = (Person) o` infers `peer` as ownership modifier for the cast to make the assignment type correct. However, we would not introduce a new cast for the assignment `x = o`. Allowing the inference to introduce additional casts would increase the risk of runtime errors significantly and defeat the purpose of static type checking.

Note, however, that Universe types support casts. In Universe types, 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 pro-

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

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

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

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

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

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

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

$$\frac{\begin{array}{cc}\vdash \overline{mpd} : \Sigma_p & env(Cid, \overline{mpd}) = \Gamma \\ \Gamma \vdash e : T, \Sigma_b & overriding(Cid, m) = \Sigma_o \\ \multicolumn{2}{c}{\Sigma = \Sigma_b \cup \Sigma_p \cup \Sigma_o \cup \{om(T) <: om(T_r), decl(om(T_r))\}}\end{array}}{Cid \vdash p\ T_r\ m(\overline{mpd})\ \{\ e\ \} : \Sigma}$$

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

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

$$\frac{\Gamma \vdash e : T_0, \Sigma_0 \qquad fType(T_0, f) = T, \Sigma_1}{\Gamma \vdash e.f : T, \Sigma_0 \cup \Sigma_1}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_0 : T_0, \Sigma_0 \\ \Gamma \vdash e_1 : T_1, \Sigma_1 \\ fType(T_0, f) = T_2, \Sigma_2 \\ \Sigma = \{om(T_1) <: om(T_2), om(T_0) \neq \texttt{any}\}\end{array}}{\Gamma \vdash e_0.f{:=}e_1 : T_2, \Sigma_0 \cup \Sigma_1 \cup \Sigma_2 \cup \Sigma}$$

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

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

Figure 4: Constraint generation rules. The constraint system contains one ownership variable for each occurrence of a reference type in the program (that is, in field declarations, method signatures, casts, and object creation). These are exactly the places where the system generates a declaration constraint.

gram, but not to introduce additional casts. Information from the runtime inference of Sec. 5 might allow one to further reduce the risk of runtime errors.

Helper functions The overloaded helper functions $fType$ and $mType$ are defined as follows. Function $fType(C, f)$ yields the declared field type of field f in class C or a superclass of C. It yields only a type, but no constraints. The overloaded function $fType(u\ C, f)$ (taking a type rather than a class as first argument) determines the type of field f adapted from viewpoint $u\ C$ to `this`. It results in an adapted field type and constraints on the ownership variable u of the viewpoint and the ownership variable for the declared type.

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

Function $mType(C, m)$ yields the declared method signature of method m in class C or a superclass of C. The overloaded 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 constraints on the ownership variable u of the viewpoint and the ownership variables for the declared parameter and return types,

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

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

respectively.

$$mType(u\ C, m) = p\ u'_r\ C_r\ m(u'_p\ C_p\ pid), \Sigma$$
where
$$mType(C, m) = p\ u_r\ C_r\ m(u_p\ C_p\ pid)$$
$$\Sigma = \{(u \rhd u_r = u'_r), (u \rhd u_p = u'_p), decl(u'_r), decl(u'_p)\}$$

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 of the constrain generation and encoding.

Class `C` contains two reference types, one in the declaration of field `f` and one in the `new` expression. For these types, the algorithm introduces ownership variables α_1 and α_2, respectively. The inference creates the following constraints:

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

Figure 6: For each kind of constraint (see Sec. 4.2), the formula that encodes it. Each constraint variable α_i corresponds to three boolean variables β_i^{rep}, β_i^{peer}, and β_i^{any}.

1. $decl(\alpha_1), decl(\alpha_2)$: we generate the declaration constraints for the two ownership variables in the program
2. $\alpha_2 \neq$ `any`: variable α_2 is a legal modifier for an object creation
3. $\alpha_2 <: \alpha_1$: variable α_2 is a subtype of variable α_1 and, thus, the assignment to field `f` is type correct.

4.3 Encoding for a SAT Solver

Once we have the constraint system Σ, it needs to be solved. We 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.

This section explains how to encode the constraints Σ as boolean formulas. These formulas are then converted to conjunctive normal form, which is the input format of the SAT solver. The SAT solver either returns an assignment of booleans that satisfies the formula or notifies the user that the formula is un-satisfiable. The assignment of booleans corresponds to ownership modifiers for the variables that satisfy all constraints.

Our implementation supports changing the solver or the encoding of the constraints, which facilitates experimentation. Fig. 6 defines the encoding of the constraints that our implementation currently uses.

Declaration ($decl(\alpha)$): We use three boolean variables β^{peer}, β^{rep}, and β^{any} to represent an ownership variable α from the constraints. Our encoding expresses that exactly one of these three booleans is assigned true.

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 [20], that is, always allowing only exactly one of the booleans to be true. Moreover, this encoding makes it simpler to express heuristics as weights; for instance, the preference for the modifier `rep` can be easily expressed by assigning a high weight to the boolean variable β^{rep}.

Subtype ($\alpha_1 <: \alpha_2$): If the subtype is `any`, then the supertype must be `any`. If the supertype is `peer`, then the subtype must be `peer`. If the supertype is `rep`, then the subtype must be `rep`.

Adaptation ($\alpha_1 \rhd \alpha_2 = \alpha_3$): The encoding of adaptation constraints reflects the definition of viewpoint adaptation given in Sec. 2. The first two lines cover the situations where viewpoint adaptation can determine the owner of an object statically. In all other cases, the result of the adaptation is `any`.

Equality & Inequality ($\alpha_1 = u$, $\alpha_1 \neq u$): These constraints are simply encoded by forcing that the corresponding boolean is either true or false.

Comparable ($\alpha_1 <:> \alpha_2$): The clauses forbid that one variable is assigned `peer` when the other variable is assigned `rep`.

4.4 Heuristic Choice of a Solution

In the simple example of Fig. 5, our system creates a constraint system with six boolean variables (for the two ownership variables) and clauses encoding the four constraints described above. The SAT solver may return any of the following four solutions to this constraint system, depending on its search strategy:

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

Like for any program, these solutions include one that assigns `peer` to all variables. This is not useful (unless it is the only possibility), because it corresponds to a completely flat ownership structure.

When choosing among many possibilities to assign ownership modifiers, a human programmer is influenced by a variety of design considerations.

- A deeper ownership structure gives better encapsulation, so it is generally preferable, but it limits sharing.

- The types in method signatures influence what clients may call the method, so it is preferable for method parameters to have the `any` modifier.
- Weights can also be used to handle other guidance, from a user or tool, more flexibly. Suppose that a user has partially annotated a program. If the annotations lead to unsatisfiable constraints, they can be converted from absolute equality constraints (as described in Sec. 4.2) into preferences. This gives the inference tool the flexibility to override annotations when necessary.
- Other heuristics are possible; for instance, a verification tool based on ownership such as Spec# [29] might have different needs.

To reflect these design considerations, our approach uses the weight feature of a weighted Max-SAT solver to encode preferable ownership typings. These solvers permit a user to express a preference for each clause to be true or false as a weight. A maximal-weight assignment that satisfies all constraints is not only correct, but is more likely to be what a human developer would want.

Our tool combines multiple heuristics, by scaling and then adding their weights. Among others, users can select and combine the following heuristics:

- We provide pre-defined general heuristics that maximize the depth of ownership structures and the applicability of methods, for example, by prefering deep ownership for fields and object creations, but `any` for method parameters.
- Program-specific heuristics can be obtained from the runtime inference as described in the next section.

Our tool allows programmers to weight the weights and scaling of these different heuristics. A user can enable/disable heuristics either before running the tool, or after observing a non-desirable tool output. However, we expect that in most practical applications, the heuristics and scaling are pre-defined or determined automatically by the tool, for instance, depending on the code coverage of the runtime inference. Therefore, programmers influence the outcome of the inference mainly by providing partial annotations and by running the runtime inference. To avoid bias, our experiments use only the built-in heuristics.

However, the heuristic for preferring deeper ownership structures over flatter structures 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. Suppose that the weights are set such that $w_{rep} > w_{peer} > w_{any}$. The maximum-weight assignment is the flat structure in Fig. 7b, which has weight $2w_{rep} + 2w_{peer}$. A programmer would most likely prefer the deeper structure in Fig. 7c, because it encapsulates the `Node` object inside the list, while allowing the list to control modifications of the nodes and maintain invariants over the node structure. However, its weight is less: $2w_{rep} + w_{peer} + w_{any}$.

The next section explains how runtime inference can be used to determine the deepest possible ownership structure. While the 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.

5. Runtime Inference

The runtime inference of Universe types obtains tracing information from one or several executions of the program. Then, it performs the following five steps (explained in Secs. 5.1–5.5) to compute the ownership modifiers:

1. Build the representation of the object store, which contains all objects created during the program executions together with information about references and modifications between them.
2. Build the dominator tree to identify for each object the candidates for its owner.
3. Resolve conflicts with the Universe type system to obtain an ownership structure that is compatible with the owner-as-modifier discipline enforced by Universe types.
4. Harmonize different instantiations of a class to obtain *static* ownership information for each reference type in the program.
5. Output ownership modifiers as described in Sec. 3, together with information about the achieve code coverage.

We illustrate the runtime inference using the classes in Fig. 8a.

5.1 Build the Representation of the Object Store

The program trace contains a record of each field reference that occurred at run time. These references are of two types. *Write references* were used to update a field or call a non-pure method on an object; these references determine the ownership structure of an application. Recall that Universe types enforce the owner-as-modifier ownership discipline, and does not restrict references in general (unlike other ownership type systems). *Naming references* were used for reading fields and calling pure methods; these are needed to map the inference results back to the source code.

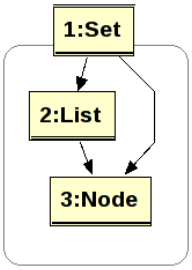
From the trace, the runtime inference builds a cumulative representation of the object store called the *Extended Object Graph* (EOG) [45, 48]. The EOG represents all objects that ever existed in the store, all references between these objects that were ever observed, and, in particular, which objects modified which other objects.

The nodes of the EOG are (representations of) run-time objects, consisting of the object's class and a unique identifier. The edges of the EOG are of two types: write edges and naming edges. (We sometimes call them "write references"

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

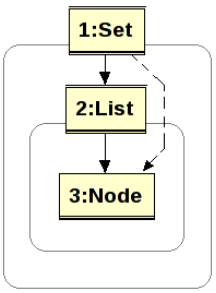
(a) Unannotated source code

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



(b) Flat ownership structure

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

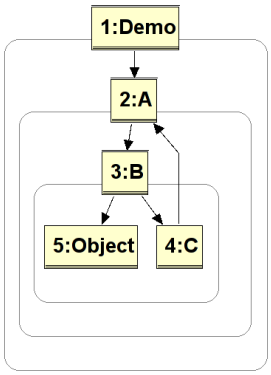


(c) Deep ownership structure

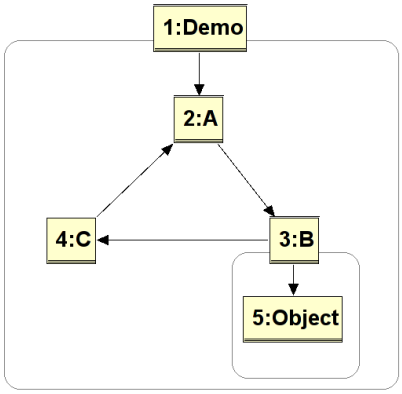
Figure 7: The static heuristics alone do not guarantee the deepest ownership structure. In this code, the `lastUsed` field implements a cache that speeds up repeated membership queries. Parts (b) and (c), show two different possible ownership structures. Contexts are depicted by rounded rectangles, and owner objects sit atop the context of objects they own. Arrows represent references and the dashed reference is signifies a reference used only for reading.

```
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(); } }
}
```

(a) Source code



(b) Dominator tree of write edges



(c) After conflict resolution

Figure 8: Running example to illustrate runtime inference.

and "naming references", when no confusion with the trace can arise.) Each naming edge is labeled with a field name.

For example, a call x`.setFoo(`y`)` introduces two edges in the EOG. A write reference from the current receiver object `this` to x represents that `this` modifies x by calling the non-pure method `setFoo`. 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.

Example In our running example (Fig. 8a), class `A` contains in its constructor the statement `b = new B(this)`. At 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. The *object creation* causes insertion, into the EOG, of 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. The *field update* causes insertion, into the EOG, of a write edge 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. All naming references for a field can later be used to infer the ownership modifier for that field.

5.2 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 [30, 4] of x.

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.

The result of finding the dominators for the example from Fig. 8a is shown in Fig. 8b. 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.

5.3 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 allows write references within a context and from an owner to an owned object. However, a dominator graph may have references (edges) from an object to an object in an enclosing context. Such write references are not permitted in the Universe type system. If such a *conflicting* edge is found in the EOG, the objects connected by the conflicting edge are raised to a common level. This process is repeated until no more conflicts are present.

Example This situation is illustrated by the code in Fig. 8a. 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. 8b 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 for Universe types 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. 8c. 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.

Algorithm The algorithm works as follows:

We assign to each object its depth in the EOG. The depth of a write reference from x to y is the minimum of the depths of its origin x and target y. A *conflict reference* is a write reference that is not permitted by the Universe type system. There are two kinds of conflict references: (1) *upward conflicts* whose origin's depth is strictly greater than the depth of the target and (2) *downward conflicts* whose origin's depth is strictly smaller than the depth of the target and whose origin does not dominate the target. Downward conflicts do not occur in the initial EOG that is obtained from the computation of the dominator tree, but may be introduced during conflict resolution.

We set up an initial priority queue that contains all conflict references in the EOG, sorted by depth. As long as the priority queue is non-empty, we select the minimum-depth reference, say a reference from x to y. We resolve this conflict by raising the object with the greater depth to the context of the object with the smaller depth. That is, for upward conflicts, we raise object x, and for downward conflicts, we raise object y. We then add to the worklist all write references to or from the raised object that have now become conflict references, and then start the next iteration.

This algorithm obviously all conflicts because it iterates until there are no conflict references in the EOG, and because each new conflict is added to the priority queue and processed. So to show that the algorithm is correct, we only have to argue that it terminates. A simple termination argument is that the sum of the depths of all nodes in the EOG is non-negative and decreases in each iteration because the origin or target of a conflict edge is raised, that is, its depth decreases. The existence of such a termination measure is a sufficient condition for termination.

An important property of our algorithm is that it raises each object at most once and, therefore, is efficient. This is achieved by processing the references in the worklist in a top-down way, that is, by starting with the reference with the smallest depth, say d. Raising the origin or target x of this reference to depth d may lead to additional conflicts. If a new conflict is an upward conflict, the conflict reference points to x and, thus, also has depth d; if a new conflict is a

downward conflict, it points from x to an object with a depth greater that d and, thus, also has depth d. Consequently, when the algorithm processes references of depth d, it will never add references to the worklist with a depth smaller than d. Therefore, no upward conflict in the worklist can have x as its origin and no downward conflict can have x as its target. Therefore, x will not be raised again.

5.4 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. If there are no casts in the program that the static inference could use to refine ownership information (see Sec. 4.2), it will override the suggestions from the runtime inference to eliminate the need for a cast.

The harmonization step is not only used to harmonize the objects within the EOG for one program execution, but also to harmonize EOGs from multiple program executions. So to support runtime inference using multiple executions as mentioned in Sec. 3, one can store the EOG of each execution after the harmonization step, that is, when each EOG represents an ownership structure that can be typed with Universe types. When several of these EOGs are combined, one has to perform another harmonization step to resolve discrepancies between the different graphs.

Example Imagine that the example program is executed twice, with and without command line arguments. Therefore, 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. 8b (which is legal because the reference from object 4 to object 2 is a naming reference) and once with a flat structure as in Fig. 8c (where the reference from object 4 to object 2 is a naming reference and, thus, object 4 has been raised to the context of object 2 during conflict resolution). The annotation for field `b` in class `A` is once `rep` and once `peer`. The harmonization algorithm then decides to use `peer` as annotation for field `b` and raises the non-conforming instance to a higher level.

5.5 Output Ownership Modifiers

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.

Types for 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. Assume for example that the runtime inference on the example in Fig. 8a was based on a single program execution, in which method `off` is not called. For such an execution, the reference from object 4 to object 2 in Fig. 8b is a naming reference such that the runtime inference determines the modifier `any` for field `a` in class `C`. However, this modifier contradicts the inequality constraint generated for the call to the non-pure method `a.off()`, which requires the receiver expression `a` to have a modifier different from `any`.

To handle such situations, the static inference treats the input from the runtime inference as *suggestions*, not as fixed annotations. So instead of generating equality constraints for the modifiers suggested by the runtime inference, the static inference just adjusts the weights for the boolean variables in the encoding. In our example above, the static inference would assign a weight of, say 50 to β^{any} and 10 each to β^{rep} and β^{peer} for `a`'s modifier. These weights will make the SAT solver *prefer* solutions where `a`'s modifier is `any`, but also allow it to find other solutions if `any` is not possible (like in our example) or if those other solutions are preferable because they have a higher overall weight.

How trustworthy the suggestions from the runtime inference are depends on the code coverage. Therefore, our runtime inference provides coverage information that is used by the static inference to determine the absolute weights of the boolean variables and also to scale the weights from the runtime inference when combining them with other heuristics as described in Sec. 4.4.

5.6 Discussion

Both the static and the runtime inference can operate on arbitrary program fragments, such as a class and its unit tests. To infer meaningful ownership modifiers, it is often neces-

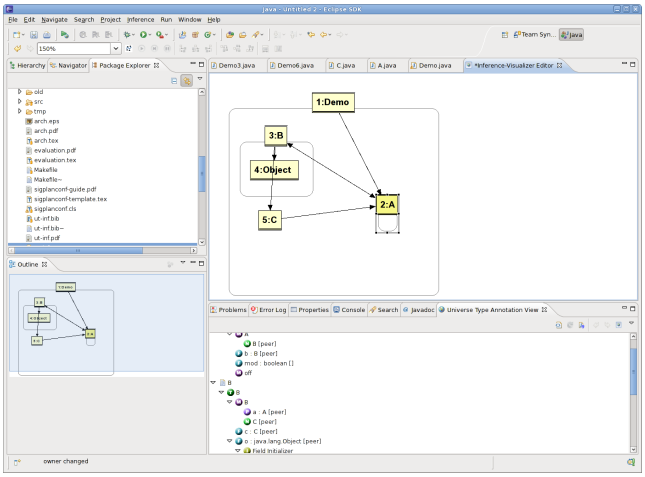


Figure 9: A screenshot showing a runtime object graph on top and the static inference on the bottom. The visualizer displays the object graph while it is built up and modified by the runtime inference. This helps programmers understand both the target program and the runtime inference algorithm. Note that the graphs in Figs. 7 and 8 are screenshots from this tool.

sary to consider larger contexts, in particular, the clients of a class. Even without such context, 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. The static inference encodes 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.

6. Experience

This section describes the implementation and our initial experience with our inference approach.

Implementation We implemented our inference approach as a set of command-line tools and also provide Eclipse plug-ins to ease the interaction for programmers. The implementation of the whole tool-chain consists of around 50,000 non-comment lines of Java and C++ source code. Fig. 9 shows a screenshot of the Eclipse plug-in.

We support the workflow as described in Sec. 3. The programmer can first determine the purity of methods; then experiment with the runtime inference and use the visualization to understand the ownership structure; then use the static inference to ensure consistent annotation of the whole program; and finally, use the annotation tool to insert the ownership modifiers into the source code.

The annotations inferred by the different tools are stored in XML files, which facilitates the comparison of multiple runs of the inference tools. The XML representation also decouples the inference from the concrete UTS syntax, which is useful to support alternative syntax such as the existing JML2 Universe syntax in backwards-compatible Java comments and the Java 7 annotations [19].

The runtime inference obtains program traces via the Java Virtual Machine Tooling Interface (JVMTI). The object graphs obtained from these traces are stored in XML files. The object graphs for multiple program runs can be combined to improve code coverage. Moreover, interactive or long-running programs need to be traced only once for each desired code path. This object graph can then be reused later without requiring human interaction or recomputing results.

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.

Experience We have used our inference tools in the course of development on many examples that test certain aspects and corner cases. The tools are simple to use and interactive work is possible with them.

One example we evaluated is a singly-linked list with iterator. The original implementation resulted in a flat ownership structure, because the iterator directly modified the list nodes. This direct modification prevents a deep ownership structure. Once we rewrote the code to forward modifications from an iterator to the corresponding list, our inference tools inferred a deep ownership structure. By applying the tool to more test cases, the inferred ownership structures improved. The final results are then inserted into the source code as JML annotations and the correctness of the annotations can be verified by the UTS type checker that is built into the JML2 checker.

To assess how well the inference algorithms scale to solve systems of constraints with many variables, we performed inference on four micro-benchmarks. The results are presented in Fig. 10.

“Independent fields” contains 5,000 fields, each assigned with a different object. This benchmark creates many small objects at runtime, highlighting the runtime performance. The static constraints are all independent from each other, stressing this kind of constraint system.

“Chained fields” has 5,000 fields; each field is assigned from the previous one, except the first that is assigned with an object. Only one object is created at runtime; the static constraints are all interdependent.

Benchmark	SLOC	Static Inference Timing (s)			CNF Size		Runtime Inference Timing (s)		
		Parse	Gen	Solve	Vars	Clauses	Orig	Trace	Solve
1. Independent fields	5,010	2.54	0.62	2.16	30006	70013	0.19	50.78	3.22
2. Chained fields	5,005	2.60	0.32	1.55	15009	40019	0.18	22.69	1.52
3. Chained objects	5,016	3.03	0.63	2.25	30015	85035	0.18	31.43	3.24
4. Conflicts	17	0.99	0.05	0.25	27	70	0.14	2.42	4.18

Figure 10: Timing results for the Micro-Benchmarks. SLOC gives the number of non-blank, non-comment lines. Column 'Parse' gives the time needed for the JML compiler to build the AST, 'Gen' the time to build the constraint system, and 'Solve' the time needed for the SAT solver to give a solution. Column 'CNF Vars' gives the number of variables and 'CNF Clauses' gives the number of clauses in the CNF encoding.

"Chained objects" has 5,000 objects, each of which has a field that points to the next object; in addition, another object has 5,000 fields and points to all the objects.

"Conflicts" has 5,000 objects, each of which has a field that points to the next object and a field that points to the root object. At runtime the program initially creates a deep ownership structure, but then the many back pointers cause the structure to collapse. The source code for this benchmark is a lot simpler than for the other benchmarks, which makes the static inference fast. Our explanation for the radically lower tracing overhead is that the Just-in-Time compiler has a chance to optimize the execution of this benchmark, whereas the previous three benchmark did not contain loops.

We executed each of these benchmarks on the command line to gather the timing information. We also inspected the benchmarks in the Eclipse plug-ins and ensured that the inferred annotations express the desired structure. We also fixed certain annotations in the solution and let the static inference propagate our constraints. Finally, we inserted the inferred ownership modifiers into the source code and used the JML2 checker to ensure that the type rules of the Universe type system are correctly followed.

Future Work Our experiments indicate that the inference tools are usable. For both runtime and static inference, the design goals were simplicity and modularity, not performance. We have made no attempt to optimize their performance, so there are many opportunities to speed up the tools.

The static inference suffers from two large overheads. The JML2 compiler is very slow (the "parsing" numbers in Fig. 10), and replacing it would also make our inference system support more Java language features. The SAT solver is invoked as a separate process, with input/output files. Invoking it in the same process would be faster.

The runtime inference spends most of its time writing trace files, in a verbose XML format. It could use a binary format, buffering, or other optimizations.

We then plan to apply our inference approach to realistic code bases and evaluate the usability with real programmers.

7. Related Work

7.1 Static inference

Milanova [34] 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 [18].

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.

The box model [43] 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.

General type qualifier inference [24] could be applied to ownership, but it would not phelp in the inference of the deepest or most desirable ownership structure. Rather, it would infer any solution that satisfies all constraints, possibly a flat structure.

Several other static analyses are tangentially related. Ma and Foster [32] present a static analysis to infer uniqueness and ownership, without mapping the results to a type system. Moelius and Souter's static analysis for ownership types resulted in a large number of ownership parameters [27]. Kacheck/J [25] infers package-level encapsulation properties. SafeJava [7, 9, 8] 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.

The system whose implementation is most similar to ours is a type inference systems against races [22]. It builds a constraint system, uses a SAT solver to find solutions, and exploits 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.

7.2 Dynamic inference

Alisdair Wren's work on inferring ownership [48] 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 [11, 5, 7, 12] 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.

Our tool contains a simple visualizer to help programmers understand and evaluate the inferred annotations. Hill et al. [26] have developed a sophisticated visualization tool based on ownership types. 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 [41] focuses on the treatment of aliasing in heap visualizations.

Rayside et al. [46] present a dynamic analysis that infers ownership and sharing, but they do not map the results back to an ownership type system. Mitchell [35] 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 [36] 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 [21] 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.

7.3 Other sources of guidance for inference

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.

8. 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) [16]. 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 [13, 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, 12, 44]. 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 [10].

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 annota-

tions for program understanding. In *Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 311–330. ACM Press, 2002.

[7] C. Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, MIT, 2004.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Principles of programming languages (POPL)*, pages 207–212. ACM Press, 1982.

[16] W. Dietl. *Universe Types: Topology, Encapsulation, Genericity, and Tools*. PhD thesis, Department of Computer Science, ETH Zurich, 2009.

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

[18] W. Dietl and P. Müller. Runtime Universe Type Inference. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2007.

[19] M. D. Ernst. Type annotations specification (JSR 308). `http://types.cs.washington.edu/jsr308/`, September 12, 2008.

[20] 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.

[21] 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.

[22] C. Flanagan and S. N. Freund. Type inference against races. In *SAS*, pages 116–132. Springer-Verlag, 2004.

[23] D. Graf. Implementing purity and side effect analysis for Java programs. Semester Project, Department of Computer Science, ETH Zurich, Winter 2005/06.

[24] 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.

[25] 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.

[26] T. Hill, J. Noble, and J. Potter. Visualizing the structure of object-oriented systems. In *IEEE International Symposium on Visual Languages (VL)*, page 191. IEEE Computer Society, 2000.

[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] 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.

[33] J. Marques-Silva. Practical applications of boolean satisfiability. In *Workshop on Discrete Event Systems (WODES)*. IEEE Press, May 2008.

[34] A. Milanova. Static inference of Universe Types. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2008.

[35] 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.

[36] 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.

[37] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002.

[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] S. Nägeli. Ownership in design patterns. Master's thesis, Department of Computer Science, ETH Zurich, 2006.

[41] J. Noble. Visualising objects: Abstraction, encapsulation, aliasing, and ownership. In *Software Visualization*, volume 2269 of *LNCS*, pages 607–610. Springer-Verlag, 2002.

[42] J. Palsberg. Type-based analysis and applications. In *Program Analysis for Software Tools and Engineering (PASTE)*, pages 20–27, 2001.

[43] 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.

[44] 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.

[45] J. Potter, J. Noble, and D. Clarke. The ins and outs of objects. In *Australian Software Engineering Conference*, pages 80–89. IEEE Press, 1998.

[46] 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.

[47] 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.

[48] A. Wren. Inferring ownership. Master's thesis, Department of Computing, Imperial College, June 2003. `http://www.cl.cam.ac.uk/~aw345/`.