Tunable Static Inference for Generic Universe Types

Werner Dietl[1], Michael D. 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, including program verification, thread synchronization, and memory management. However, the annotation overhead of ownership type systems hampers their widespread application. This paper addresses this issue by presenting a tunable static type inference for Generic Universe Types. In contrast to classical type systems, ownership types have no single most general typing. Our inference chooses among the legal typings via heuristics. Our inference is tunable: users can indicate a preference for certain typings by adjusting the heuristics or by supplying partial annotations for the program. We present how the constraints of Generic Universe Types can be encoded as a boolean satisfiability (SAT) problem and how a weighted Max-SAT solver finds a correct Universe typing that optimizes the weights. We implemented the static inference tool, applied our inference tool to four real-world applications, and inferred interesting ownership structures.

1 Introduction

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

Object ownership [10] 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 [6]. 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 [33]. To obtain these benefits, 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 adoption of ownership type systems. Standard techniques for static type inference [12] are not applicable. First, there is no need to check for the existence of a correct typing; a flat ownership structure gives a trivial typing. 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 static inference for Generic Universe Types [14, 13], a lightweight ownership type system designed to enable program verification [32]. 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, through a runtime analysis, or through interaction with the programmer.

The main contributions of this paper are:

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

Tunable Inference: use of heuristics and programmer interaction to indicate which among many legal solutions is preferable; this approach is implemented by use of a weighted Max-SAT solver.

Evaluation: an implementation of our inference scheme on top of the OpenJDK compiler, and an illustration of its effectiveness on real programs.

This paper is organized as follows. Sec. 2 gives background on Generic Universe Types. Sec. 3 overviews the inference system using examples. Sec. 4 formalizes the static inference, consisting of the core programming language, the constraint generation rules, the weighting heuristics, and the encoding as a weighted SAT problem. Sec. 5 describes our implementation and our experience with it. Finally, Sec. 6 discusses related work, and Sec. 7 concludes.

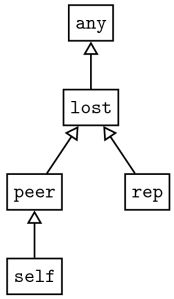
2 Background on Generic Universe Types

Generic Universe Types (GUT) [14, 13] is an ownership type system that allows programmers to describe and enforce hierarchical heap topologies and optionally enforces the owner-as-modifier encapsulation discipline based on the topology. GUT is integrated into the tool suite of the Java Modeling Language (JML) [23].

```
public class Person {
  peer Person spouse;
  rep  Account savings;
  rep  List<peer Person> friends;

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

(a) Example program.



(b) Ownership modifier type hierarchy.

Fig. 1. (a) A simple example of Generic Universe Types. A `Person` object owns its savings account and has the same owner as its spouse. It also owns a `List` of `Person` objects, each of which is its peer. (b) The type hierarchy of the ownership modifiers; see Sec. 2 for an explanation.

Ownership Topology. GUT organizes the heap hierarchically into contexts and restricts modifications across context boundaries. As in most other ownership systems, each object has at most one owner object. The ownership relation is acyclic. A *context* is the set of objects sharing an owner.

In GUT, a programmer expresses the ownership topology by writing one of three *ownership modifiers* on each reference type. An ownership modifier expresses ownership relative to the current receiver object `this`.

`peer` expresses that the referenced object is in the same context as the current object `this`. For example, in Fig. 1a, a `Person p` has the same owner as `p.spouse`.

`rep` expresses that the referenced object is owned by the current object. For example, in Fig. 1a, a `Person p` is the owner of `p.savings`.

`any` gives no static information about the relationship of the two objects.

In addition, the formalization uses two internal ownership modifiers, which are not part of the surface syntax:

`lost` expresses that the two objects have a relationship, but that relationship is not expressible as `peer` or `rep`. For example, in Fig. 1a, `spouse.savings` is a "nephew" of `this`; GUT cannot express this relationship, so it gives `spouse.savings` the ownership modifier `lost`.

`self` is used only for the current receiver object `this`.

Fig. 1b gives the type hierarchy. A `self`-modified type is a subtype of the corresponding `peer` type because `self` denotes the `this` object, which is obviously a peer of `this`. Types with `self`, `rep`, and `peer` modifiers are subtypes of the corresponding type with a `lost` modifier because `lost` conveys less ownership information. Similarly, an `any`-modified type is a supertype of all other versions.

The example in Fig. 1a also illustrates the use of ownership modifiers with generic types. Field `friends` has type `rep List<peer Person>`, which expresses that the `List` object is owned by the `Person` object containing the field, whereas the elements stored in the list are peers of that object. Note that the ownership modifier of a type argument is interpreted relative to the client that instantiates the generic type (here, the `Person` object), not the object of the generic type.

Compound expressions: viewpoint adaptation and `lost`*.* The modifier of a compound expression is determined by combining the ownership modifiers of its components. For example, consider a field access `tony.spouse`, where `tony` is of type `rep Person`. This expression traverses first a `rep` reference and then a `peer` reference, so its modifier is the result of adapting `tony`'s `spouse` modifier from the viewpoint of `tony` (where it is `peer`) to the viewpoint of `this`. Here, this adaptation yields `rep` because the resulting object has the same owner as `tony`, which is `this`.

In some cases, this *viewpoint adaptation* leads to a loss of static ownership information. For example, the expression `spouse.savings` traverses first a `peer` and then a `rep` field, so the resulting object has a specific relationship to `this`, but the relationship cannot be expressed in the type system. GUT uses a special ownership modifier `lost` to express this. Two different expressions of `lost` type might stand for different unknown relationships, so it would be illegal to assign one `lost` expression to another one. GUT remains sound by prohibiting the `lost` type on the left-hand side of an assignment. This explains why GUT introduces `lost` rather than reusing `any` to stand for an unknown relationship: it would be too restrictive to forbid all assignments to left-hand-sides of type `any`.

Formally, viewpoint adaptation is a function $\triangleright$ that takes two ownership modifiers and yields the adapted modifier. (1) $\texttt{peer} \triangleright \texttt{peer} = \texttt{peer}$; (2) $\texttt{rep} \triangleright \texttt{peer} = \texttt{rep}$; (3) $u \triangleright \texttt{any} = \texttt{any}$; (4) $\texttt{self} \triangleright u = u$; and (5) for all other combinations the result is `lost`. In Fig. 1a, the modifier of `spouse.savings` is $\texttt{peer} \triangleright \texttt{rep} = \texttt{lost}$. Since `lost Person` is a subtype of `any Person`, the expression may be assigned to variable `a`.

In addition to field accesses, viewpoint adaptation also occurs for parameter passing, result passing, and type variable bound checks.

Encapsulation. Generic Universe Types enforce that programs adhere to the heap topology described by the ownership modifiers. In addition, they optionally enforce an encapsulation scheme called the *owner-as-modifier* discipline [15]: 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 enforce the invariant $\texttt{savings.balance} \geq 0$ because the owner-as-modifier discipline prevents aliases to the `Account` object `savings` from modifying 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 a `lost` or `any` reference. For

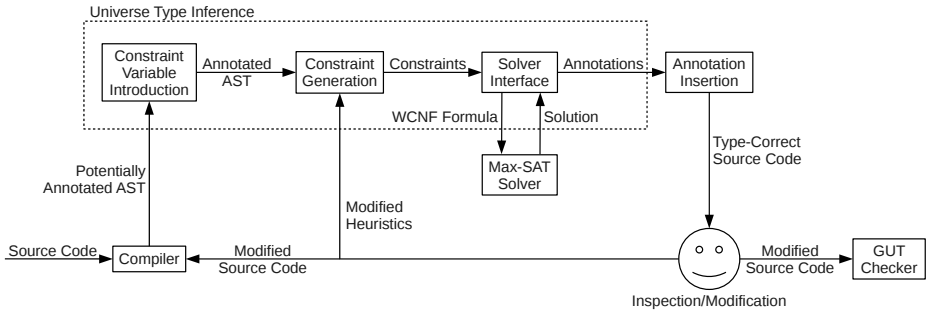


Fig. 2. Overview of the inference approach. See Sec. 3 for a detailed discussion.

instance, the call `spouse.savings.withdraw(1000)` is rejected by the type system because the viewpoint-adapted modifier of the receiver, `spouse.savings`, is `lost`. A `lost` or `any` reference can still be used for field accesses and to call pure (side-effect-free) methods. For instance, method `assets` in Fig. 1a may read the `balance` field via the `any` reference `a`.

Because the default modifier is `peer`[3], an un-annotated Java program is a legally-typed program in GUT. This typing describes a flat ownership structure — no object is owned by any other object — and so it imposes no constraints on, nor guarantees about, the program's operation. Therefore, inference is needed to automatically produce annotations that express a deeper ownership structure.

3 Inference Approach and Example

Given a Java program as input, which may be partially annotated with ownership modifiers, the static inference determines a legal GUT typing. Fig. 3 shows an example input program and four inferred typings. Fig. 2 overviews the process. Sec. 3.1 discusses the type inference process (the dotted rectangle of Fig. 2), which is the focus of this paper. Sec. 3.2 explains how a user can iteratively use our toolset (the rest of Fig. 2).

3.1 Inference Approach

Type inference has three main steps: creating constraint variables, creating constraints over those variables, and solving the constraints to infer a typing.

The inference first creates a constraint variable for each possible occurrence of an ownership modifier or of viewpoint adaptation in the source code. Recall that GUT allows ownership modifiers for all reference types. Primitive types and type variable declarations/uses do not take ownership modifiers, and therefore no constraint variables are created for them. However, the upper bound of a type variable takes ownership modifiers. In the example in Fig. 3, a total of 14

[3] There are a few exceptions. For instance, subtypes of `Throwable` have the `any` modifier by default to allow the propagation of exceptions across ownership contexts.

```
class Person {
  Person1 spouse;
  Account2 savings;
  List3<Person4>12 friends;

  void marry(Person5 p) {
    spouse = p;
  }

  void befriend(Person6 p) {
    friends.add(p);
  }

  int assets() {
    Account7 a = spouse.savings13;
    return savings.balance + a.balance;
  }

  void demo() {
    Person8 o1 = new Person9();
    Person10 o2 = new Person11();
    this.marry(o1);
    o1.befriend(o2)14;
  }
}
```

Solution	α_1	α_2	α_3	α_4	α_5	α_6	α_7	α_8	α_9	α_{10}	α_{11}	α_{12}	α_{13}	α_{14}
no weights	peer	peer	peer	peer	peer	peer	peer	peer	peer	peer	peer	any	peer	peer
default	any	rep	rep	any	any	any	any	rep	rep	rep	rep	any	lost	any
alternative	rep	rep	rep	peer	rep	peer	any	rep	rep	rep	rep	any	lost	rep
manual	*peer*	rep	rep	any	peer	any	any	peer	peer	rep	rep	any	lost	any

Fig. 3. An example un-annotated program and typings. Our algorithm creates constraint variables α_1–α_{11} corresponding to locations 1–11 of type uses, and creates α_{12}–α_{14} corresponding to the viewpoint adaptations induced by the expressions at locations 12–14. The figure also shows four inference solutions that are inferred by our tool, depending on the heuristics used and whether the programmer provides a partial annotation. In the last solution, the italic `peer` modifier for α_1 was manually added to the source code before inferring the solution.

constraint variables are introduced: α_1–α_{11} correspond to the locations where ownership annotations may appear in the source code and α_{12}–α_{14} are constraint variables for viewpoint adaptation. For example, a constraint variable α_1 is introduced for the ownership modifier in the `spouse` field type, and constraint variable α_9 is introduced for the ownership modifier in the `new` expression. One generic type contains multiple constraint variables corresponding to the main and type argument modifiers; for the type of field `friends`, α_3 is introduced for the main modifier and α_4 is introduced for the type argument. Viewpoint adaptation introduces additional constraint variables, for example, α_{12} represents the result of adapting the declared upper bound of the type variable of class `List` (assumed to be `any Object`) from the point-of-view of `friends` to `this`.

The inference generates constraints over the constraint variables by traversing the AST. These constraints correspond one-to-one to the type rules of GUT [13]. Additional, weighted, breakable constraints express preferences regarding the solution, obtained by applying a heuristic. For Fig. 3, the tool generates a total of 35 constraints. For example, the assignment `spouse = p` results in a subtype constraint between constraint variables α_5 and α_1. Constraints are also generated to connect the constraint variables involved in viewpoint adaptation. For example, the field access `spouse.savings` has to adapt the declared type of field `savings` from the point-of-view of `spouse` to `this`. To model the result

of this adaptation, the additional constraint variable α_{13} was introduced. The constraint $\alpha_1 \rhd \alpha_2 = \alpha_{13}$ is generated to encode the dependency between the involved variables. The assignment to local variable `a` then induces a subtype constraint between α_{13} and α_7.

The constraints are translated into a weighted SAT formula. A weighted Max-SAT solver [5] finds a solution that satisfies all of the type system constraints and the breakable constraints resulting in the maximum weight. The SAT solution is translated into a typing for the program: a concrete ownership modifier for each constraint variable in the program. Any such typing is guaranteed to be correct, as all type rules are encoded as mandatory constraints; our experiments also confirm this.

3.2 Iterative Usage

As explained in the introduction, the best typing in an ownership type system depends on the programmer's intent and how the annotations will be used by downstream tools. Therefore, we expect the inference tool to be used iteratively.

Using the inference without heuristics results in the first solution presented in Fig. 3: all modifiers are assigned `peer`, except for α_{12} which is the result of adapting the declared `any` upper bound. This flat assignment is usually not the desired solution.

By using our tool's built-in heuristics (see Sec. 4.3), we get the second result in Fig. 3. This heuristic prefers a deep ownership structure and broadly applicable methods. For example, this heuristic prefers α_5 and α_6 to be `any`, making the methods callable with arbitrary arguments, even though this solution requires α_1 and α_4 to be `any`, reducing the encapsulation guarantees of these fields. α_2 and α_3 are inferred to be `rep`, as they are not dependent on parameter types.

If the inferred annotations do not reflect the programmer's design intent, the programmer can improve the result in three ways: customize the heuristics, manually add annotations, or fix defects in the source code.

(1) The programmer may customize the heuristics to encourage 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. By contrast, when the whole program is available, the programmer might select heuristics that favor restrictive modifiers to facilitate subsequent use of the ownership information, for instance by a program verifier. In our example, the third solution was inferred using a heuristic that prefers `rep` as annotation for fields and has no other weights. Note how in this solution α_4 is inferred to be `peer`. A `rep` solution is not possible in this location, as it would result in α_6 to be `rep`, which would make the call `o1.befriend(o2)` impossible. On the other hand, α_5 was inferred to be `rep`; this was possible, because the only call of `marry` is on a `this` receiver. Calls of `marry` with a receiver other than `this` are impossible in this solution. This changed heuristic provides stronger encapsulation, but limits the future use of fields and methods. In Sec. 4.3 we will discuss weighting in more detail.

(2) The programmer can write ownership modifiers in the input program. The inference system always respects such annotations, even if the program text

would have permitted another solution. In our example, the fourth solution is the result of manually adding a `peer` annotation to field `spouse`, fixing constraint variable α_1 to `peer`. Using our default heuristics again, this would result in the parameter to `marry` also becoming `peer`. Also the type of `o1` needs to be inferred to be `peer` in order for `this` and `o1` to be peers in the call of `marry`.

(3) The programmer might fix defects in the source code. One variety of defect is programmer-written type annotations that are incompatible with one another or with usage by the source code. The inference system points out such errors (currently by providing no inference result; improved error messages are future work). In our example, assume the programmer manually adds a `peer` annotation to field `spouse` and a `rep` annotation to parameter `p` of method `marry`; no assignment to the remaining constraint variables could resolve the mismatch of modifiers in the assignment `spouse = p`.

A more subtle defect is one that prevents inference of the programmer's intended ownership structure, even though the system outputs some legal typing of the program. For example, suppose that a programmer designed a layered system in which lower layers do not call non-pure methods of higher layers. A violation of this property would cause the inference to produce a flatter-than-desired ownership structure. The programmer could correct the source code so that it implements the design.

Once the programmer is satisfied with the 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 during program maintenance, and to make them available to downstream tools such as a program verifier. The GUT type checker can be used to ensure that the annotations remain consistent.

4 Tunable Static Inference

This section formalizes our inference approach. Sec. 4.1 presents a core calculus for a Java-like programming language, which is used by the rest of the formalism. Sec. 4.2 gives syntax-directed type inference rules: each programming language construct gives rise to a set of constraint variables and to a set of constraints over the variables. We introduce a constraint variable for each location in the source program where a concrete ownership modifier may be written and for each expression that requires viewpoint adaptation. Any solution to the constraints is a legal assignment of a concrete ownership modifier to each source location. Sec. 4.3 describes how to add additional constraints that express preferences among the possible solutions. Finally, Sec. 4.4 shows how to encode all the constraints as a weighted SAT problem, and to transform a weighted Max-SAT solver's output into a set of concrete ownership modifiers for the program.

4.1 Programming Language

Fig. 4 summarizes the syntax of the language and the naming conventions.

$$
\begin{array}{rcl}
P & ::= & \overline{Cls} \\
Cls & ::= & \texttt{class}\ Cid\langle\overline{TP}\rangle\ \texttt{extends}\ C\langle\overline{T}\rangle\ \{\ \overline{fd}\ \overline{md}\ \} \\
TP & ::= & X\ \texttt{extends}\ N \\
md & ::= & p\ \langle\overline{TP}\rangle\ T_r\ m(\overline{T\ pid})\ \{\ e\ \} \\
e & ::= & \texttt{null} \mid x \mid \texttt{new}\ N() \mid e.f \mid e_0.f{:=}e_1 \mid \\
 & & \quad e_0.\langle\overline{T}\rangle m(\overline{e}) \mid (N)\ e \\
u & ::= & \alpha \mid \texttt{peer} \mid \texttt{rep} \mid \texttt{any} \mid \texttt{lost} \mid \texttt{self}
\end{array}
\qquad
\begin{array}{rcl}
C & ::= & Cid \mid \texttt{Object} \\
fd & ::= & T\ f; \\
p & ::= & \texttt{pure} \mid \texttt{impure} \\
T & ::= & N \mid X \\
N & ::= & u\ C\langle\overline{T}\rangle \\
x & ::= & pid \mid \texttt{this}
\end{array}
$$

pid	parameter identifier	f	field identifier
m	method identifier	Cid	class identifier
α	constraint variable identifier	X	type variable identifier

Fig. 4. Syntax of our programming language. A sequence of A elements is denoted as $\overline{A}$. The surface syntax (written by the programmer) does not include ownership modifiers α, `lost`, or `self`, and allows omitting ownership modifiers. The only difference from previous formalizations of GUT [13] is the addition of constraint variables α as a placeholder for a concrete ownership modifier.

A program P consists of a sequence of class declarations; P is implicitly available in all judgments. A class declaration *Cls* names the class and its superclass, along with their type parameters and type arguments, respectively, and gives field and method declarations. A field declaration is a simple pair of a type and an identifier. A method declaration consists of the method purity, method type parameters if any, return type, method name, formal parameter declarations, 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 either a non-variable type N or a type variable X. A non-variable type N consists of an ownership modifier u and a possibly-parameterized class C. The definition of the ownership modifiers is the only deviation from previous formalizations of GUT. Ownership or Universe modifiers u include the *concrete ownership modifiers* `peer`, `rep`, `any`, `lost`, and `self`, as well as constraint variables α. Constraint variables α are used as placeholders for the concrete ownership modifiers that the system will infer. The surface syntax does not include α, `lost`, or `self` and allows omitting ownership modifiers; constraint variables are introduced for all omitted ownership modifiers.

4.2 Building the Constraints

This section introduces constraint variables (Sec. 4.2.1), the kinds of constraints (Sec. 4.2.2), and the syntax-directed rules that build the constraints (Sec. 4.2.3).

4.2.1 Constraint variables. A *constraint variable* represents the ownership modifier for the occurrence of a reference type or a particular expression.

For each position where a concrete ownership modifier may occur in the solution — that is, for each use of a type — our tool introduces a constraint

variable α that represents the ownership modifier for that position. Our inference will later assign one of the concrete ownership modifier `rep`, `peer`, or `any` to each of these constraint variables. The tool also introduces a constraint variable for each expression that induces viewpoint adaptation; these will be assigned to `rep`, `peer`, `any`, or `lost`. `self` is used only as the type of the `this` literal and never inferred.

To infer the ownership modifiers for the program of Fig. 3, our tool would introduce a constraint variable for each numbered location.

If the programmer has partially annotated the program, then the generated constraints use the programmer-written modifier instead of creating a constraint variable.

4.2.2 Constraints. The inference rules (Sec. 4.2.3) create five kinds of constraints over the ownership modifiers, in particular, over the constraint variables.

Subtype ($u_1 <: u_2$): A subtype constraint enforces that u_1 will be assigned an ownership modifier that is a subtype of the ownership modifier assigned to u_2. Subtype constraints are used for assignments and for pseudo-assignments (parameter passing, result passing, type variable bound checks).

Adaptation ($u_1 \triangleright u_2 = \alpha_3$): An adaptation constraint ensures that the viewpoint adaptation of variable u_2 from the viewpoint expressed by u_1 results in α_3.

Equality ($u_1 = u_2$): An equality constraint ensures that two modifiers are the same. They are used to handle method overriding and type argument subtyping, which are both invariant.

Inequality ($u_1 \neq u_2$): An inequality constraint ensures that two modifiers differ. For example, the type system forbids the `lost` modifier on the left-hand side of an assignment. The type system also forbids the `any` modifier for the receiver of field updates, if the owner-as-modifier discipline is enforced.

Comparable ($u_1 <:> u_2$): 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 for casts.

Fig. 6 in Sec. 4.2.3 defines helper judgments that lift these constraints from ownership modifiers to types.

4.2.3 Constraint generation. Our system takes as input a program and creates a set Σ of the kinds of constraints defined in Sec. 4.2.2. The constraints in Σ are satisfied by any correct GUT typing for the program. The constraints correspond to the type rules [13] expressed abstractly over the ownership modifiers.

Fig. 5 contains the rules for extracting constraints from a program. It defines judgments over class, field, and method declarations, as well as over expressions. Our inference is a type-based analysis [36] that runs only on valid Java programs. Therefore, our rules do not encode all Java type rules, but give only constraints for the additional checks for Generic Universe Types. To simplify the notation,

the rules use helper judgments and functions that lift constraints from single ownership modifiers to types; they are defined in Figs. 6 and explained after the discussion of the main judgments.

An environment Γ maps type variables of the enclosing class and method to their upper bounds and variables to their types. We use the notation $\Gamma(X)$ and $\Gamma(x)$ to look-up the upper bound of a type variable and the type of a variable, respectively. Helper function env (defined in Fig. 6) defines the environment necessary for checking class and method declarations.

We now discuss the rules of Fig. 5.

The constraints for a *class, field, method parameter, and method declaration* consist of the constraints for their components. The well-formedness of types is ensured using the well-formed type (OK) judgment defined in Fig. 6. For a method declaration, note that the environment is extended with the method type variables and the method parameters. Function overriding requires that, if the current method is overriding a method in a superclass, the parameter and return types are consistent. The resulting constraint set Σ_2 defines equality constraints between the types in the current method signature and a directly overridden method signature. For space reasons, we do not show the formal definition of overriding; it follows directly from the GUT formalization [13].

Finally, there are eight judgments for *expressions*, which are also mostly standard. We discuss casts immediately below. For an object creation expression the main ownership modifier has to be different from `lost` and `any` to ensure that either `peer` or `rep` is inferred, giving the new object a specific location in the ownership topology. Helper functions fType and mType, discussed below, yield the field type, respectively the method signature, after viewpoint adaptation, and additional constraints that encode the necessary adaptations of modifiers. To ensure soundness of the inferred results, `lost` has to be forbidden for all types involved in pseudo-assignments: the adapted field type, and the adapted method parameter types and method type variable bounds. These constraints ensure that modifications are only possible if the ownership is known statically. The rules for a field update and for an impure method call generate additional constraints only when the owner-as-modifier discipline is enforced: the main modifier of the receiver expression has to be different from `lost` and `any` to ensure that the owner of the modified object is statically known.

The $\Gamma \vdash N <:> T_0 : \Sigma_c$ clause of the cast rule requires explanation. Recall that a cast is a type loophole that indicates that the program's behavior is beyond the reasoning capabilities of the type system. If the un-annotated input program contains a cast, then the corresponding runtime check might fail at run time. Generic Universe Types also support casts: downcasts that specialize ownership information (that is, casts from `any` to `peer` or `rep`) and require a runtime check. Our inference never inserts a new cast; to do so would defeat the purpose of static ownership type checking. However, the inference is permitted to choose arbitrary[4] ownership modifiers at existing casts, and therefore an existing cast

[4] Actually, the choice is not arbitrary. The <:> constraint requires the two types are comparable — otherwise, the cast is guaranteed to fail.

Environment: $\Gamma = \{\overline{X \mapsto N}; \overline{x \mapsto T}\}$

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

$$\frac{\begin{array}{c}\mathrm{env}(Cid, \overline{TP}) = \Gamma \qquad \Gamma \vdash \overline{fd} : \Sigma_f \qquad \Gamma \vdash \overline{md} : \Sigma_m \\ \Gamma \vdash \texttt{self}\ C\langle\overline{T}\rangle, \mathrm{bounds}(\overline{TP})\ \mathrm{OK} : \Sigma_t \\ \Sigma = \Sigma_f \cup \Sigma_m \cup \Sigma_t\end{array}}{\vdash \texttt{class}\ Cid\langle\overline{TP}\rangle\ \texttt{extends}\ C\langle\overline{T}\rangle\ \{\ \overline{fd}\ \overline{md}\ \} : \Sigma}$$

Field and method parameter declaration: $\boxed{\Gamma \vdash T\ f : \Sigma}$, $\boxed{\Gamma \vdash T\ pid : \Sigma}$

$$\frac{\Gamma \vdash T\ \mathrm{OK} : \Sigma}{\Gamma \vdash T\ f : \Sigma} \qquad \frac{\Gamma \vdash T\ \mathrm{OK} : \Sigma}{\Gamma \vdash T\ pid : \Sigma}$$

Method declaration: $\boxed{\Gamma \vdash md : \Sigma}$

$$\frac{\begin{array}{c}\mathrm{env}(\Gamma, \overline{TP}, \overline{T\ pid}) = \Gamma' \qquad \Gamma' \vdash \overline{T\ pid} : \Sigma_0 \\ \Gamma' \vdash e : T, \Sigma_1 \qquad \mathrm{overriding}(\Gamma', m) = \Sigma_2 \\ \Gamma' \vdash \mathrm{bounds}(\overline{TP}), T_r\ \mathrm{OK} : \Sigma_3 \\ \Gamma' \vdash T <: T_r : \Sigma_4\end{array}}{\Gamma \vdash p\ \langle\overline{TP}\rangle\ T_r\ m(\overline{T\ pid})\ \{\ e\ \} : \bigcup_{i=0}^{i=4} \Sigma_i}$$

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

$$\frac{\Gamma \vdash e : T_0 : \Sigma_0 \qquad \Gamma \vdash T_0 <: T : \Sigma_1}{\Gamma \vdash e : T, \Sigma_0 \cup \Sigma_1} \qquad \frac{}{\Gamma \vdash \texttt{null} : T, \emptyset} \qquad \frac{}{\Gamma \vdash x : \Gamma(x), \emptyset} \qquad \frac{\begin{array}{c}\Gamma \vdash e_0 : T_0, \Sigma_0 \\ \Gamma \vdash N\ \mathrm{OK} : \Sigma_t \\ \Gamma \vdash N <:> T_0 : \Sigma_c \\ \Sigma = \Sigma_0 \cup \Sigma_t \cup \Sigma_c\end{array}}{\Gamma \vdash (N)\ e_0 : N, \Sigma}$$

$$\frac{\Gamma \vdash N\ \mathrm{OK} : \Sigma_0 \qquad \Sigma_1 = \{\mathrm{om}(N) \neq \{\texttt{lost}, \texttt{any}\}\}}{\Gamma \vdash \texttt{new}\ N() : N, \Sigma_0 \cup \Sigma_1} \qquad \frac{\begin{array}{c}\Gamma \vdash e_0 : N_0, \Sigma_0 \qquad \Gamma \vdash e_1 : T_1, \Sigma_1 \\ \mathrm{fType}(N_0, f) = T_2, \Sigma_2 \qquad \Gamma \vdash T_1 <: T_2 : \Sigma_3 \\ \Sigma_4 = \{\texttt{lost} \notin T_2\} \\ \Sigma_5 = \{\boxed{\mathrm{om}(N_0) \neq \{\texttt{lost}, \texttt{any}\}}\}\end{array}}{\Gamma \vdash e_0.f := e_1 : T_2, \bigcup_{i=0}^{i=5} \Sigma_i}$$

$$\frac{\Gamma \vdash e : N_0, \Sigma_0 \qquad \mathrm{fType}(N_0, f) = T, \Sigma_1}{\Gamma \vdash e.f : T, \Sigma_0 \cup \Sigma_1} \qquad \frac{\begin{array}{c}\Gamma \vdash e_0 : N_0, \Sigma_0 \qquad \Gamma \vdash \overline{e_a} : \overline{T_a}, \Sigma_1 \\ \mathrm{mType}(N_0, m, \overline{T}) = p\ \langle\overline{TP}\rangle\ T_r\ m(\overline{T_p\ pid}), \Sigma_2 \\ \Gamma \vdash \overline{T_a} <: \overline{T_p} : \Sigma_3 \\ \Sigma_4 = \{\texttt{lost} \notin (\overline{T_p}, \mathrm{bounds}(\overline{TP}))\} \\ \Gamma \vdash \overline{T}\ \mathrm{OK} : \Sigma_5 \qquad \Gamma \vdash \overline{T} <: \mathrm{bounds}(\overline{TP}) : \Sigma_6 \\ p = \texttt{impure} \Rightarrow \Sigma_7 = \{\boxed{\mathrm{om}(N_0) \neq \{\texttt{lost}, \texttt{any}\}}\} \\ p = \texttt{pure} \Rightarrow \Sigma_7 = \emptyset\end{array}}{\Gamma \vdash e_0.\langle\overline{T}\rangle m(\overline{e_a}) : T_r, \bigcup_{i=0}^{i=7} \Sigma_i}$$

Fig. 5. Constraint generation rules. Helper judgments and functions are defined in Figs. 6. The generated constraint set Σ encodes all constraints that need to be fulfilled to give a valid GUT program. The two framed constraints only need to be generated if the owner-as-modifier discipline should be enforced.

might fail either because of the base language check, or because of the ownership check. 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. Our choice is a natural extension of the base language behavior. An alternative would be for the static inference to choose modifiers in such a way as to guarantee that the runtime ownership check at each cast succeeds. This can be accomplished by simply changing "<:>" to "=". Subsumption of the expression type could then still be used to cast to a supertype, which is guaranteed to succeed.

Helper judgments and functions. Fig. 6 defines additional judgments and functions that support the main ones of Fig. 5.

Function om gives the main ownership modifier for a non-variable type. Function bounds gives the upper bound types from type parameter declarations. We compact the notation to compare one ownership modifier against a set of ownership modifiers and to ensure that an ownership modifier does not appear in a type. Function env defines the environment depending on the surrounding class and method declarations.

For space reasons, we omit showing how each judgment is also lifted to sequences of elements by applying the judgment to the individual elements and combining the results.

Viewpoint adaptation is lifted from single modifiers (defined in Sec. 4.2.2) to types using two judgments: (1) adapting a type from an ownership modifier and (2) adapting a type from the viewpoint of a non-variable type. A type is adapted from the viewpoint of an ownership modifier to `this`, giving an adapted type and a constraint set. There are two cases. No constraint is generated to adapt type variables X, as they do not need to be adapted. The constraints to adapt a non-variable type $u'\ C\langle\overline{T}\rangle$ from viewpoint u consist of the constraint for combining u with the main modifier u', resulting in a fresh constraint variable α, and recursively adapting the type arguments. A type is adapted from the viewpoint of a non-variable type to `this`, by first adapting the type using the main modifier u and then substituting the type arguments $\overline{T}$ for the type variables $\overline{X}$. Function typeVars gives the type variables defined by a class. The notation $T[\,\overline{T}/\overline{X}\,]$ is used to substitute type arguments $\overline{T}$ for occurrences of type variables $\overline{X}$ in T.

The subtyping judgment between two types determines a constraint set that has to hold in order for the two types to be subtypes. The most interesting subtyping rule is the second one, which derives a subtyping relationship from a subclassing relationship by adapting the type arguments from the superclass to the particular subtype instantiation. The subclassing relationship $\sqsubseteq$ is the reflexive and transitive closure of the `extends` relationship of the classes; it is defined over instantiated classes $C\langle\overline{T}\rangle$, as defined in GUT [13].

Fig. 6 does not show the lifted versions of equality and comparable constraints. The equality constraint is lifted to types by simple recursion. A comparable constraint is applied to two non-variable types by first going to a common

Notation:

$\text{om}(u\ C\langle\overline{T}\rangle) \equiv u$ $\qquad (u \neq \{u_1, u_2, \ldots\}) \equiv (u \neq u_1, u \neq u_2, \ldots)$

$\text{bounds}(\overline{X\ \texttt{extends}\ N}) \equiv \overline{N}$ $\qquad (u \notin u'\ C\langle\overline{T}\rangle) \equiv (u \neq u' \wedge u \notin \overline{T})$

Environment definitions:

$\text{env}(Cid, \overline{X\ \texttt{extends}\ N}) = \{\overline{X \mapsto N}; \texttt{this} \mapsto \texttt{self}\ Cid\langle\overline{X}\rangle\}$

$\text{env}(\{\overline{X_c \mapsto N_c}; \overline{x \mapsto T}\}, \overline{X\ \texttt{extends}\ N}, \overline{T_p\ pid}) = \{\overline{X_c \mapsto N_c}, \overline{X \mapsto N}; \overline{x \mapsto T}, \overline{pid \mapsto T_p}\}$

Ownership modifier - type adaptation: $\boxed{u \rhd T = T' : \Sigma}$

$$\frac{}{u \rhd X = X : \emptyset} \qquad \frac{\Sigma_0 = \{u \rhd u' = \alpha\} \quad \text{fresh}(\alpha) \quad u \rhd \overline{T} = \overline{T'} : \Sigma_1}{u \rhd u'\ C\langle\overline{T}\rangle = \alpha\ C\langle\overline{T'}\rangle : \Sigma_0 \cup \Sigma_1}$$

Type - type adaptation: $\boxed{\Gamma \vdash N \rhd T = T' : \Sigma}$

$$\frac{u \rhd T = T_1 : \Sigma \quad T_1\left[\,\overline{T}/\overline{X}\,\right] = T' \quad \text{typeVars}(C) = \overline{X}}{u\ C\langle\overline{T}\rangle \rhd T = T' : \Sigma}$$

Subtyping: $\boxed{\Gamma \vdash T <: T' : \Sigma}$

$$\frac{\Sigma = \{u <: u', \overline{T} = \overline{T}'\}}{\Gamma \vdash u\ C\langle\overline{T}\rangle <: u'\ C\langle\overline{T'}\rangle : \Sigma} \qquad \frac{C\langle\overline{X}\rangle \sqsubseteq C'\langle\overline{T_1}\rangle \quad u\ C\langle\overline{T}\rangle \rhd \overline{T_1} = \overline{T'}, \Sigma}{\Gamma \vdash u\ C\langle\overline{T}\rangle <: u\ C'\langle\overline{T'}\rangle : \Sigma}$$

$$\frac{}{\Gamma \vdash X <: X : \emptyset} \qquad \frac{}{\Gamma \vdash X <: \Gamma(X) : \emptyset} \qquad \frac{\Gamma \vdash T <: T_1 : \Sigma_1 \quad \Gamma \vdash T_1 <: T' : \Sigma_2}{\Gamma \vdash T <: T' : \Sigma_1 \cup \Sigma_2}$$

Well-formed type: $\boxed{\Gamma \vdash T\ \text{OK} : \Sigma}$

$$\frac{\Gamma \vdash \overline{T}\ \text{OK} : \Sigma_0 \quad \text{typeBounds}(u\ C\langle\overline{T}\rangle) = \overline{T'}, \Sigma_1 \quad \Gamma \vdash \overline{T} <: \overline{T'} : \Sigma_2}{\Gamma \vdash u\ C\langle\overline{T}\rangle\ \text{OK} : \bigcup_{i=0}^{i=2} \Sigma_i} \qquad \frac{X \in \Gamma}{\Gamma \vdash X\ \text{OK} : \emptyset}$$

Adaptation of a field type: $\boxed{\text{fType}(N, f) = T, \Sigma}$

$\text{fType}(u\ C\langle\overline{T}\rangle, f) = T', \Sigma$ where $\text{fType}(C, f) = T$, $u\ C\langle\overline{T}\rangle \rhd T = T' : \Sigma$

Adaptation of a method signature: $\boxed{\text{mType}(N, m, \overline{T}) = p\ \overline{TP}\ T_r\ m(\overline{T_p\ pid}), \Sigma}$

$\text{mType}(N, m, \overline{T'}) = p\ \langle\overline{X\ \texttt{extends}\ N'}\rangle\ T'_r\ m(\overline{T'_p\ pid}), \Sigma_b \cup \Sigma_r \cup \Sigma_p$

where $N = u\ C\langle\overline{T}\rangle$, $\text{mType}(C, m) = p\ \langle\overline{X\ \texttt{extends}\ N_b}\rangle\ T_r\ m(T_p\ pid)$, $N \rhd \overline{N_b} = \overline{N_0} : \Sigma_b$, $N \rhd T_r = T_{r0} : \Sigma_r$, $N \rhd \overline{T_p} = \overline{T_{p0}} : \Sigma_p$, $\overline{N_0}[\overline{T'}/\overline{X}] = \overline{N'}$, $T_{r0}[\overline{T'}/\overline{X}] = T'_r$, $\overline{T_{p0}}[\overline{T'}/\overline{X}] = \overline{T'_p}$

Adaptation of type bounds: $\boxed{\text{typeBounds}(N) = \overline{N'}, \Sigma}$

$\text{typeBounds}(u\ C\langle\overline{T}\rangle) = \overline{N'}, \Sigma$ where $\texttt{class}\ C\langle\overline{X\ \texttt{extends}\ N}\rangle \ldots \in P$, $u\ C\langle\overline{T}\rangle \rhd \overline{N} = \overline{N'}, \Sigma$

Look-up of class type variables: $\boxed{\text{typeVars}(N) = \overline{X}}$

$\text{typeVars}(u\ C\langle\overline{T}\rangle) = \overline{X}$ where $\texttt{class}\ C\langle\overline{X\ \texttt{extends}\ N}\rangle \ldots \in P$

Fig. 6. Helper judgments and functions for the constraint generation rules of Fig. 5.

superclass and then generating a comparable constraint for the two main modifiers and equality constraints for the type arguments.

The well-formed type (OK) judgment defines when a type T is well-formed in an environment Γ giving the constraints Σ. We omit judgments for well-formedness of environments, which are basically just well-formedness for all involved types.

The overloaded helper functions fType, mType, and typeBounds 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(N, f) (taking a non-variable type rather than a class as first argument) determines the type of field f adapted from viewpoint N to `this`. It results in an adapted field type and constraints on the constraint variables of the viewpoint and the constraint variables for the declared type.

Function mType(C, m) yields the declared method signature of method m in class C or a superclass of C. The overloaded function mType$(N, m, \overline{T})$ determines the method signature of method m adapted from viewpoint N to `this` and substituting method type arguments for their type variables. It results in an adapted method signature and constraints on the constraint variables of the viewpoint and the constraint variables for the declared parameter, return, and type variable bound types, respectively.

Function typeBounds$(u\ C\langle\overline{T}\rangle)$ yields the upper bounds of the type variables of class C adapted from the non-variable type $u\ C\langle\overline{T}\rangle$ to `this` and a set of constraints.

4.3 Heuristic Choice of a Solution

For a given set of constraints, the solver may return any satisfying assignment. For completely un-annotated programs, these solutions include the trivial one that assigns `peer` to all variables. It is typically not the desired solution 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.
- Other heuristics are possible; for instance, a verification tool based on ownership, such as Spec# [24], has different needs for invariants and pre-/post-conditions.

To reflect these design considerations, we attach weights to some constraints. All constraints of the type system are mandatory. For each constraint variable, we use the position of the variable in the AST to encode a preference for a particular solution by adding an additional breakable, weighted equality constraint.

For variables α that appear in ...:

- ... field types, the weight for $\alpha = \texttt{rep}$ is 80.
- ... parameter types, the weight for $\alpha = \texttt{any}$ is 150.
- ... return types, the weight for $\alpha = \texttt{rep}$ is 30.
- ... class and method type variable bounds, the weight for $\alpha = \texttt{any}$ is 200.

This pre-defined heuristic prefers solutions with deep ownership structures and generally applicable methods. A user may adapt these weights, either globally or for individual variables. In the example from Fig. 3, the "alternative" solution was generated using 100 as weight for `rep` for variables that appear in field types and using no other weights. As discussed previously, this alternative weighting results in stronger encapsulation at the cost of the applicability of the methods. In the rest of this paper we will use the pre-defined heuristics; examining the effect of alternative weights is interesting future work.

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. The annotations are encoded as mandatory equality constraints. If the partial annotations lead to unsatisfiable constraints, the tool could—after consulting the developer—convert them from mandatory into breakable constraints. This would give the inference tool the flexibility to override annotations when necessary. The programmer would then inspect what annotations needed to be changed.

4.4 Encoding for a SAT Solver

Once the constraint system Σ is generated, it needs to be solved. We encode Σ as a weighted Max-SAT problem and use an existing solver [5] for three reasons. First, Generic Universe Types allow only a fixed number of 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 unsatisfiable. The assignment of booleans corresponds to ownership modifiers for the variables that satisfy all constraints.

We finally turn all the formulas into the CNF format used by the Max-SAT evaluation benchmarks [27]. This format is supported by different SAT solvers, and our implementation supports changing the solver.

Encoding of constraint variables. Four boolean variables β_i^{peer}, β_i^{rep}, β_i^{any}, and β_i^{lost} represent each ownership variable α_i from the constraints. The encoding expresses that exactly one of these four booleans is assigned true:

$$(\beta^{peer} \vee \beta^{rep} \vee \beta^{any} \vee \beta^{lost}) \wedge \neg(\beta^{peer} \wedge \beta^{rep}) \wedge \neg(\beta^{peer} \wedge \beta^{any}) \wedge \neg(\beta^{peer} \wedge \beta^{lost}) \wedge \neg(\beta^{rep} \wedge \beta^{any}) \wedge \neg(\beta^{rep} \wedge \beta^{lost}) \wedge \neg(\beta^{lost} \wedge \beta^{any})$$

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

Fig. 7. For each kind of constraint (see Sec. 4.2.2), the formula that encodes it. Each constraint variable α_i is encoded by four boolean variables β_i^{rep}, β_i^{peer}, β_i^{any}, and β_i^{lost}.

For every variable that will be inserted into the program, `lost` is forbidden and the encoding of the variable is accordingly simplified.

An alternative encoding would use only two booleans to encode the four possibilities. Such an encoding would have fewer variables, but more complicated clauses to encode constraints. Our encoding can be solved more efficiently [17].

Encoding of constraints. Fig. 7 defines the encoding of the constraints from the constraint set Σ into formulas over the boolean variables and follows the definitions given in Sec. 2.

We use simpler encodings when the constraint is between a variable and a concrete ownership modifier. For example, the equality constraint $\alpha_i = \texttt{peer}$ is encoded by the formula β_i^{peer}.

Encoding of weights. We use the weighting feature of a weighted Max-SAT solver to encode the weights of Sec. 4.3.

For each mandatory constraint we use the maximum weight, which the SAT solver treats as infinity; this enforces that all the type rules are fulfilled. For each breakable constraint we use the determined weight.

5 Implementation and Experience

This section describes the implementation (Sec. 5.1), our experience with it (Sec. 5.2), and possible future work (Sec. 5.3).

The implementation, experimental setup, and results are publicly available[5].

[5] `http://www.cs.washington.edu/homes/wmdietl/inference/`

Benchmark	SLOC	Constraint Size			CNF Size			Timing (seconds)			
		vars	constraints		vars	clauses		topol.		encap.	
			topol.	encap.		topol.	encap.	gen	solve	gen	solve
1. zip	2611	455	2411	2949	4656	13639	14063	4.5	1.1	4.5	1.1
2. javad	1846	364	2571	3113	4988	14989	15333	3.5	1.0	3.6	1.0
3. jdepend	2460	824	4868	6024	9752	28110	29176	5.1	1.4	5.8	1.5
4. classycle	4658	1548	8726	10242	17756	53062	54380	6.0	1.8	6.2	2.0

Fig. 8. Size and timing results. SLOC gives the number of non-blank, non-comment lines as determined by the sclc tool. The constraint size columns give the number of constraint variables and constraints in the program. The CNF size gives the number of boolean variables and clauses in the CNF encoding. Finally, the timing columns give the time for generating (gen) and solving (solve) the constraints. We executed each run three times and report the median. The number of constraints and clauses and the timing is further sub-divided into whether annotations for only the topology or also for enforcing the encapsulation discipline should be inferred. This choice does not affect the number of constraint variables or boolean variables.

5.1 Implementation.

The static inference is implemented on top of the Checker Framework [37], which is a pluggable type checking framework built on top of the JSR 308 branch of the OpenJDK compiler [16]. By building our inference tool on the OpenJDK compiler, the tool supports full Java. The implementation of many language features is significantly simplified by the Checker Framework, which provides an abstraction of a basic type checker. The annotations inferred by the tool are stored in the input format of the Annotation File Utilities (AFU), which can automatically insert the inferred annotations into the source code. The AFU format also facilitates the comparison of multiple runs of the inference tool. We separately implemented a Generic Universe Types checker that handles Java programs with GUT annotations.

The inference implementation consists of around 4400 non-comment, non-blank lines of Scala code. The biggest development effort was spent on introducing unique constraint variables and mapping them to AFU output positions. Constraint generation reuses a lot of the existing Checker Framework infrastructure.

Our tool is modular and only generates constraints for the part of the program that is supplied as input. For the remainder of the program, in particular for the JDK libraries, the tool currently uses the default modifier `peer`. The Checker Framework supports a library annotation mechanism and we plan to use this feature to provide a version of the JDK that is annotated with Generic Universe Types.

5.2 Experience.

We applied the tool to four real-world, open source tools developed by external developers. Fig. 8 presents size and timing information and Fig. 9 presents statistics of the inferred annotations.

Benchmark	Topology					Encapsulation				
	`peer`	`rep`	`any`	% `rep`	% `any`	`peer`	`rep`	`any`	% `rep`	% `any`
1. zip	306	81	68	18%	15%	322	93	40	20%	9%
2. javad	185	87	92	24%	25%	279	55	30	15%	8%
3. jdepend	529	175	120	21%	15%	600	160	64	19%	8%
4. classycle	1132	193	223	13%	14%	1165	188	195	12%	13%

Fig. 9. Number of inferred annotations, separated into inferring only the topology and also inferring the owner-as-modifier encapsulation discipline.

The four subjects are: (1) OpenJDK's implementation of the zip and gzip compression algorithms, taken from OpenJDK 7 build 138, (2) javad[6], a Java class file disassembler, (3) JDepend[7], a quality metrics tool, and (4) Classycle[8], a Java class dependency analyzer.

For each subject, we inferred a solution for the ownership topology and a solution that also enforces the owner-as-modifier encapsulation discipline. For this we manually added around 300 purity annotations to the four projects; in the future we plan to integrate an automated purity analysis.

We evaluated three qualities of our tool implementation:

1. correctness of the inferred annotations,
2. usefulness of the inferred annotations, and
3. scalability with respect to performance.

Correctness. We inserted the inferred annotations into the source code and ran the GUT type checker on these programs. In each case, the type checker verified the correctness of the inference results.

The inference and type checker are independent implementations which are separately implemented on top of the Checker Framework. Each is based on the proved formalization of GUT [13].

The most notable limitation is related to raw types and local inner classes. Our tool soundly infers an ownership modifier for the missing type arguments, but the external annotation tool we use—the Annotation File Utilities—does not support adding new type arguments to a raw type; code that uses raw types might not compile. We manually added such annotations in our case studies. Additionally, local inner classes are not correctly identified by the AFU annotation tool and fail to insert; we did not encounter this problem in our case studies.

Usefulness. We manually examined the inference results and believe they accurately reflect the ownership properties of the original programs. The relatively large number of `peer` annotations in Fig. 9, indicating a flat inferred ownership structure, is expected. We ran the inference tools on un-modified programs and did not attempt to improve the structure of the programs. The programs were

[6] `http://www.bearcave.com/software/java/javad/`, downloaded in December 2010.
[7] `http://www.clarkware.com/software/JDepend.html`, version 2.9.1.
[8] `http://classycle.sourceforge.net/`, version 1.3.3.

probably written with an intuitive sense of the desired ownership, but with no tools to help the programmer achieve that goal. Another limitation that causes a flat ownership structure is using the `peer` default modifier for libraries. Method parameters and upper bounds of type variables that were defaulted to `peer` force structures to be flatter than desired. It will be interesting future work to allow changing both the structure of the programs and improving the library annotations. Considering this, 10–20% inferred `rep` annotations is promising.

The number of `any` annotations consistently decreases when enforcing the owner-as-modifier discipline. It is interesting to observe that the number of `rep` references increased or decreased, depending on the application. Our explanation for this is that modifications that can be performed with an `any` receiver without enforcing an encapsulation discipline need to use a `peer` or `rep` receiver when an encapsulation discipline is enforced.

Performance. Parsing the Java files took the bulk of the inference time. Encoding the constraints into CNF, waiting for the SAT solver, and decoding the results used only around a quarter of the total time.

To experiment with scalability we applied the inference tool to JabRef[9], a bibliography management tool consisting of around 74000 SLOC. The inference generated 24402 variables and 248858 constraints, which were then translated to 521152 boolean variables and 1606319 CNF clauses. Generation of the constraint system took a total of 41 seconds and solving the system took a total of 66 seconds, of which 42 seconds were spent in the SAT solver. Unfortunately, the Annotation File Utilities crash when inserting the annotations, so we cannot use the GUT type checker to verify the results.

The used hardware was a desktop machine with two CPUs, each a 4 core Intel Xeon E5405 CPU at 2.00 GHz running Fedora 13 Linux 32 bit and using OpenJDK 7 build 138. The total main memory available is 8 GB, but the Java heap space is limited to 1 GB. The maximum observed pre-GC memory consumption during constraint generation for our four case studies was around 160MB; when run on JabRef, the maximum pre-GC consumption was 510 MB. All our software is single threaded.

5.3 Future Work.

Usability. We have shown initial results that our tool scales and produces correct results, but this is not enough: it must also be usable by and useful to real programmers. We plan to perform case studies and experiments to evaluate the quality of the inferred typings, to investigate what ownership structures occur in real programs, and to see how our tool is used in practice.

At the moment, the tool provides no information about an unsatisfied constraint system. We plan to exploit the ability of Max-SAT solvers to return a partially-fulfilling assignment, to direct the programmer towards conflicts in the system.

[9] `http://jabref.sourceforge.net`, version 2.6.

Other ownership systems. We expect that our inference approach can be adapted to other ownership type systems.

First, we plan to extend our inference to support ownership transfer [11, 34], which requires inference of the uniqueness of variables and coping with dynamic changes of ownership information.

Second, we plan to investigate how our approach can be adapted to ownership-parametric type systems [3, 10, 39]. 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.

Third, we plan to explore how we can infer ownership annotations for more complex topologies such as ownership domains [2] or multiple ownership [8].

Performance. Our inference tool seems to be fast enough for its expected use case. We have made no attempt to optimize performance, so there are many opportunities to speed up the tool, if performance becomes a problem.

Integration into an IDE would give access to the internal AST. This would cut the cost of the AST generation and allow for immediate interaction with the developer.

The SAT solver is invoked as a separate process and the input CNF is written to a file and the output from the solver needs to be parsed and interpreted. The advantage of this design is that we can use an arbitrary Max-SAT solver that supports the Max-SAT evaluation format [27]. We currently use the Sat4j solver [5] and plan to experiment with tighter integration, cutting out the file generation and parsing overheads.

6 Related Work

We discuss related work on ownership inference and on other inference.

SafeJava [6, 7] provides intra-procedural ownership type inference for local variables to reduce the annotation overhead. Agarwal and Stoller [1] describe a runtime technique that infers further annotations. In contrast, we provide a static analysis that infers all necessary annotations in a program.

AliasJava [3] combines ownership and aliasing. It uses a constraint system to infer alias annotations. A key difference to our work is that the inference for AliasJava needs to introduce ownership parameters for classes, whereas GUT expresses ownership via ownership modifiers. The inference for AliasJava potentially results in classes with many more ownership parameters than what would have been used by a programmer. The system does not support partial annotations to fix the number of ownership parameters. In contrast, GUT allows one to associate ownership modifiers with type arguments for existing type parameters, but the inference never needs to introduce additional type parameters.

The box model [38] separates the program into module interfaces and implementations. Ownership annotations are still required for the module interface, but are automatically inferred for the implementations.

Pedigree types [25] 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. Our approach also handles generic types.

Milanova [28] presented a static inference of ownership annotations, then together with Vitek extended the work to the owner-as-dominator system [29, 30]. Their tool constructs a static approximation to the object graph via an alias analysis and then computes dominators to obtain candidates for owners. Milanova et al.'s work differs from ours in five major aspects. (1) They do not use one of the extant ownership type systems, nor do they formally define the meaning of the inferred annotations. This makes it difficult to compare the precision of the tools. Moreover, the correctness of the inferred annotations can only be checked manually. Our work is based on the formalization of GUT, which has been proved sound, and we used the GUT type checker to confirm the correctness of the inferred annotations. (2) Their tool infers only annotations for field declarations and `new` expressions, whereas we infer all annotations required by Generic Universe Types. For example, for the zip/gzip programs, their tool outputs 81 annotations whereas ours outputs 455. (3) Milanova et al. use a whole-program pointer analysis, whereas our approach is modular and thus applicable to libraries and single classes. (4) Even though Milanova et al.'s analysis is asymptotically faster ($O(n^2)$ versus the NP-completeness of SAT), our tool seems to outperform theirs in practice. For example, their tool took 27 and 28 seconds to analyze the gzip and zip programs, respectively, whereas ours took only 5.6 seconds for both programs together (for comparison, their experiments were performed on a MacBook Pro with unspecified CPU). (5) Milanova et al.'s analysis does not handle generics, whereas ours does.

Ma and Foster [26] present a static analysis that combines an intraprocedural points-to analysis and an interprocedural predicate inference to infer uniqueness and ownership properties. Their system uses a strict definition of ownership and found less than 2% of parameters to be owned; it also does not map the results to a type system.

Kacheck/J [20] infers package-level encapsulation properties. The system extracts a set of boolean constraints from a bytecode program. These constraints encode that a class is not confined (that is, its objects may be accessed outside the package that contains the class), that a method is not anynomous (that is, it potentially assigns `this` to a non-confined type), and implications between these properties. The constraint system is a set of ground Horn clauses, which is solved in linear time. The solution indicates which classes are confined. Kacheck/J and our system share the goal of inferring encapsulation properties via constraint solving, but differ in three important aspects. (1) For confined types, there is a best solution, namely the one with the largest set of confined types, whereas our system uses weights and a Max-SAT solver to compute desirable solutions. (2) Confined

types support a non-hierarchical topology of static contexts, whereas GUT offers hierarchies of contexts that can be created dynamically. The enforcement of an encapsulation policy is optional in our system. (3) Confined types do not support generics, but our inference does.

Moelius and Souter's static analysis for ownership types resulted in a large number of ownership parameters [31].

Baker [4] observes that Hindley-Milner type inference uses distinct datatype nodes to represent disjoint runtime values. Therefore, the information computed by this inference can be used to infer aliasing information. O'Callahan and Jackson's Lackwit tool [35] uses this idea to compute aliasing information for C programs. Lackwit associates a tag with each type constructor in a program and then uses Hindley-Milner type inference to compute equalities between these tags. Variables whose types have different tags cannot be aliases. The alias information computed by Lackwit is useful for various software engineering tasks, but not sufficient to infer ownership. In particular, Lackwit cannot distinguish several instances of the same data structure, for instance, to infer whether the nodes of two list instances may be shared. Guo et al. [21] show how to perform a similar analysis dynamically, increasing precision.

General type qualifier inference [9, 19] infers any solution that satisfies all constraints, which is not useful for ownership types, where a trivial solution always exists.

Transitioning from a non-generic to a generic program [22] also deals with an under-constrained type inference problem that uses heuristics to determine a good solution.

The system whose implementation is most similar to ours is a type inference system against races [18]. 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 among satisfiable solutions.

7 Conclusion

We presented a novel approach to static ownership inference. To the best of our knowledge, each of the following points is unique to our system. (1) Our system accommodates preferences among multiple legal typings; the preferences can come from sources such as heuristics and partial annotations. (2) It uses a Max-SAT solver to encode programmer preferences and produce a desirable inference result. (3) Our system infers ownership types for an existing, formally-defined type system that supports generic types. (4) It infers complete ownership type annotations for realistic programs. (5) The system supports either only inferring an ownership topology, or also enforcing the owner-as-modifier encapsulation discipline. (6) Its results are both correct, as verified by a type checker, and desirable, as verified by manual inspection.

Acknowledgments. We thank the reviewers for their extensive feedback. Werner Dietl was supported in part by a fellowship from the Swiss National Science Foundation (SNSF). This work was also supported by NSF grant CNS-0855252.

References

1. R. Agarwal and S. D. Stoller. Type Inference for Parameterized Race-Free Java. In *VMCAI*, volume 2937 of *LNCS*, pages 149–160, 2004.
2. J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In *ECOOP*, volume 3086 of *LNCS*, pages 1–25, 2004.
3. J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *OOPSLA*, pages 311–330, 2002.
4. H. G. Baker. Unify and conquer (garbage, updating, aliasing, ...) in functional languages. In *LISP and functional programming (LFP)*, pages 218–226, 1990.
5. D. Le Berre and A. Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010. Available from `http://www.sat4j.org/`.
6. C. Boyapati. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, MIT, 2004.
7. C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA*, pages 211–230, 2002.
8. N. Cameron, S. Drossopoulou, J. Noble, and M. Smith. Multiple ownership. In *OOPSLA*, pages 441–460, 2007.
9. B. Chin, S. Markstrum, T. Millstein, and J. Palsberg. Inference of user-defined type qualifiers and qualifier rules. In *ESOP*, volume 3924 of *LNCS*, pages 264–278, 2006.
10. D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, 1998.
11. D. Clarke and T. Wrigstad. External uniqueness is unique enough. In *ECOOP*, volume 2743 of *LNCS*, 2003.
12. L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL*, pages 207–212, 1982.
13. W. Dietl. *Universe Types: Topology, Encapsulation, Genericity, and Tools*. PhD thesis, Department of Computer Science, ETH Zurich, 2009.
14. W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In *ECOOP*, volume 4609 of *LNCS*, pages 28–53, 2007.
15. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, 2005.
16. M. D. Ernst. Type annotations specification (JSR 308). Available from `http://types.cs.washington.edu/jsr308/`, September 12, 2008.
17. M. D. Ernst, T. D. Millstein, and D. S. Weld. Automatic SAT-compilation of planning problems. In *IJCAI*, pages 1169–1176, 1997.
18. C. Flanagan and S. N. Freund. Type inference against races. In *SAS*, pages 116–132, 2004.
19. D. Greenfieldboyce and J. S. Foster. Type qualifier inference for Java. In *OOPSLA*, pages 321–336, 2007.
20. C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. In *OOPSLA*, pages 241–253, 2001.

21. P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst. Dynamic inference of abstract types. In *ISSTA*, pages 255–265, 2006.
22. A. Kieżun, M. D. Ernst, F. Tip, and R. M. Fuhrer. Refactoring for parameterizing Java classes. In *ICSE*, pages 437–446, 2007.
23. 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. Available from `http://www.jmlspecs.org/`, 2008.
24. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *ECOOP*, volume 3086 of *LNCS*, pages 491–516, 2004.
25. Y. D. Liu and S. Smith. Pedigree types. In *IWACO*, 2008.
26. K. Ma and J. S. Foster. Inferring aliasing and encapsulation properties for Java. In *OOPSLA*, pages 423–440, 2007.
27. Max-SAT evaluation input and output format, February 2010. Available from `http://www.maxsat.udl.cat/10/requirements/`.
28. A. Milanova. Static inference of Universe types. In *IWACO*, 2008.
29. A. Milanova and Y. Liu. Practical static ownership inference. Technical Report RPI/DCS-09-04, Rensselaer Polytechnic Institute, March 2010.
30. A. Milanova and J. Vitek. Static dominance inference. In *TOOLS*, LNCS, 2011. To appear.
31. S. E. Moelius and A. L. Souter. An object ownership inference algorithm and its application. In *Mid-Atlantic Student Workshop on Programming Languages and Systems*, 2004.
32. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. 2002.
33. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.
34. P. Müller and A. Rudich. Ownership transfer in Universe Types. In *OOPSLA*, pages 461–478, 2007.
35. R. O'Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference. In *ICSE*, pages 338–348, 1997.
36. J. Palsberg. Type-based analysis and applications. In *PASTE*, pages 20–27, 2001.
37. M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA*, pages 201–212, 2008.
38. 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, 2007.
39. A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic ownership for generic Java. In *OOPSLA*, pages 311–324, 2006.