Runtime Universe Type Inference

Werner Dietl
ETH Zurich, Switzerland
`Werner.Dietl@inf.ethz.ch`
`http://www.sct.inf.ethz.ch/`

Peter Müller
Microsoft Research, USA
`mueller@microsoft.com`

Abstract

The Universe type system is an ownership type system for object-oriented languages that enforces the owner-as-modifier discipline. One strength of the Universe type system is its low annotation overhead. Still, annotating existing software is a considerable effort.

In this paper, we describe how we can analyze the execution of programs and infer ownership modifiers from the execution. These modifiers help to understand the organization of a system and can also be re-inserted into the original source code. This allows a programmer to enforce the maintenance of a specific ownership structure. We implemented runtime Universe type inference as a C program that traces the JVM execution, a Java application that infers the Universe annotations, and a set of Eclipse plug-ins that integrates the interaction with the other tools.

1. Introduction

The Universe type system [13] is an ownership type system for object-oriented languages that enforces the owner-as-modifier discipline. The type checker and runtime support for Universe Types are implemented in the JML tool suite [21].

At runtime, the *owner* of an object is either another object in the store or the special *root object*. Objects that share the same owner are grouped into a *context*; objects that have the root object as owner are in the *root context*. Ownership builds a tree rooted at the root object.

The owner-as-modifier discipline ensures that the owner of an object controls all modifications of an owned object, that is, only references to objects in the same context and to owned objects can be used for modifications. This discipline enables the modular verification of invariants [27].

Statically, the Universe type system uses three different *ownership modifiers* to build this ownership structure. The modifier `peer` expresses that the current object `this` is in the same context as the referenced object, the modifier `rep` expresses that the current object is the owner of the referenced object, and the modifier `any` does not give any static information about the relationship of the two objects. References with an `any` modifier convey less information as references with a `peer` or `rep` modifier with the same class and are therefore supertypes of the two more specific types.

The owner-as-modifier discipline is enforced by forbidding field updates and non-pure method calls through `any` references. An `any` reference can still be used for field accesses and to call pure methods. The method modifier `pure` is used to mark methods that leave objects in the pre-state of a method call unchanged.

A distinguishing characteristic of the Universe type system is its low annotation overhead compared to other ownership type systems. The annotation effort is further reduced by default modifiers. Reference types by default have the `peer` ownership modifier; only exceptions and immutable types default to `any`. These defaults make the conversion from Java to Universe Types simple, as all programs that do not directly modify caught exceptions continue to compile. However, these defaults only provide a flat ownership structure.

Standard techniques for static type inference [10] are not applicable. First, we do not have to check the existence of a correct typing. Such a typing trivially exists by making all ownership modifiers `peer`, that is, by having a flat ownership structure. Second, there is no notion of a best or most precise Universe typing. Usually, there are many possible typings, and it depends on the intent of the programmer which one to prefer.

In this paper, we describe how ownership modifiers for deep ownership structures can be found by *runtime inference*, that is, by observing the execution of a program. This approach does not require that the source code of the program is available. By using the dominator algorithm we ensure that the result is the deepest possible ownership structure that conforms to the Universe type rules. A deep ownership structure maximizes encapsulation and facilitates program verification. Nevertheless, it might not be what the programmer intended. The solution of our program therefore still needs to be reviewed by the programmer to ensure that it corresponds to the intended design.

Runtime inference depends on good code coverage to produce meaningful results. To achieve better coverage we use multiple program traces to infer the ownership modifiers. We also combine the results of runtime inference with our static inference tools [29, 16] to ensure that the final solution gives valid Universe Types for the complete program.

1.1 Related Work

Wren's work on inferring ownership [32] provided a theoretical basis for our work. 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 [8, 3, 7, 9] which uses parametric ownership and enforces the owner-as-dominator property. 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. No implementation is provided.

Daikon [14] is a tool to detect likely program invariants from program traces. Invariants are only enforced at the beginning and end of methods and therefore also snapshots are only taken at these spots. From these snapshots we cannot infer which references were used for reading and which were used for writing. Therefore we could not directly use Daikon, but our tool has a similar architecture. In the future we hope to apply optimizations from Daikon to our tool.

SafeJava [7] provides intra-procedural type inference and default types to reduce the annotation overhead. Agarwal and Stoller [1] describe a run-time technique that infers even more annotations. AliasJava [4] uses a constraint system to infer alias annotations.

Another static analysis for ownership types resulted in a large number of ownership parameters [19]. In contrast, by using runtime information we achieve a deep ownership structure and the simplicity of Universe Types makes the mapping to static annotations possible.

Rayside et al. [30] present a dynamic analysis that infers ownership and sharing, but does not map the result back to an ownership type system. Mitchell [26] analyzes the runtime structure of Java programs and characterizes them by their ownership patterns. The tool can work with heaps with 29 million objects and creates succinct graphs. The tool does not distinguish between read and write references and the results are not mapped to an ownership type system.

Work on the dynamic inference of abstract types [18] uses the flow of values in a program execution to infer abstract types. Yan et al. [33] use state machines to map implementation events to architecture events and thereby deduce architectures. Both approaches do not seem to be applicable to infer ownership information.

1.2 Running Example

We use the classes in Fig. 1 to illustrate how the algorithm works. This is a very simple and artificial example to illustrate all aspects of the algorithm. The main class is `Demo`; the Java entry-point `main` creates an instance of class `Demo` and calls method `testA` on that instance. The argument is a boolean that depends on the number of command line arguments. Method `testA` creates an `A` instance. Class `A` stores the boolean flag and creates an instance of class `B`. Class `B` creates a `C` instance and a `java.lang.Object` instance. Finally, class `C` stores a reference to the `A` object it receives and depending on the value of the `mod` field calls the `off` method on the `A` instance. The execution of the `main` method in class `Demo` results in the objects depicted in Fig. 2.

Outline. Sec. 2 describes the algorithm to infer ownership modifiers from runtime information, Sec. 3 gives implementation details, and Sec. 4 describes the Eclipse plug-ins. Finally, Sec. 5 discusses future work and concludes.

2. Runtime Universe Type Inference

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

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

We describe these steps in the following subsections. We discuss static methods at the end of this section.

2.1 Build the Representation of the Object Store

From a program execution we get a sequence of modifications of the object store. Instead of looking at only single snapshots of the store (as in [26]), we build a cumulative representation of the object store. This so-called *Extended Object Graph* (EOG) [32] represents all objects that ever existed in the store, all references between these objects that were ever observed, and, in particular, which objects modified which other objects. The information about modifications is particularly important since Universe Types do not restrict references in general (unlike other ownership type systems), but the modification of objects.

For each object in the EOG, we record information about its fields as well as the parameters and results of its methods. We use

```
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 {
    boolean mod;
    B b;

    A(boolean m) {
        mod = m;
        b = new B(this);
    }

    void off() {
        mod = false;
    }
}

class B {
    C c;
    Object o;

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

class C {
    A a;

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

Figure 1: Running example to illustrate our inference algorithm.

this information to infer ownership modifiers for these variables. Local variables are treated in a subsequent step as we describe in Sec. 2.5.

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

For example, a call x.`foo`(y) introduces two edges in the EOG. A write references from the current receiver object `this` to x represents that `this` modifies x by calling the non-pure method `foo`. This reference will later influence the ownership relation between `this` and x. A naming reference from x to y represents that a

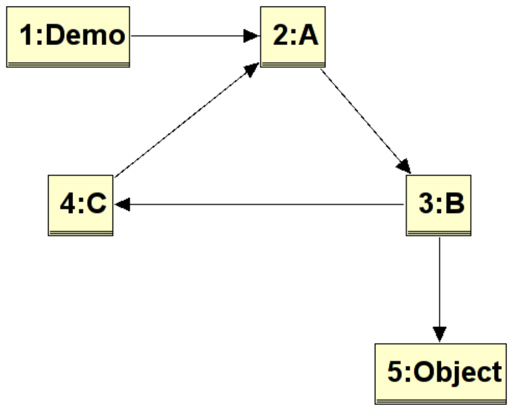


Figure 2: The store at the end of method `main` in class `Demo`. Objects are depicted by rectangles and are labeled with an identifier and the class name. References are depicted by arrows.

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.

To determine whether a method call constitutes a modification, we need purity information. We require that the purity of methods is provided as input to our tool. There are algorithms [31] to infer method purity and we also implemented a tool [17] to help with this task.

In our running example (Fig. 1), class `A` contains the statement `b = new B(this)`. On the bytecode level, this corresponds to two steps, first the creation of a new object and then the update of the field `b` of the current object. For an object creation, we insert a write edge from the current receiver object to the newly created object. In Fig. 2, this corresponds to the edge from object 2 to object 3. This write edge ensures that the ownership modifier for the object creation is either `peer` or `rep`, which is a requirement of the Universe type system. For a field update, we store a write reference from the current object to the receiver of the field update and a naming reference from the receiver of the field update to the object on the right-hand side. The naming reference is labeled with the field name. All naming references for a field can later be used to infer the correct ownership modifier for that field.

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 the array object and the objects stored in the array. For arrays, we added a special kind of naming reference that stores the relationship between the array object and the objects that are stored in the array. These references can then be used to determine the second ownership modifier.

2.2 Build the Dominator Tree

Universe Types require that all modifications of an object are initiated by its owner. For the EOG, this means that all chains of write references from the root object to an object x must go through x's owner. Therefore, we can identify suitable candidates for the owner of x by computing the dominators of x. The concept of dominators is well-known in the compiler field [2], and efficient algorithms have been developed [22].

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. They are later used to find the correct static ownership modifiers.

The result of finding the dominators for the graph from Fig. 2 is shown in Fig. 3a. 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.

2.3 Resolve Conflicts with the Universe Type System

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

This problem is illustrated by the code in Fig. 1. 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. 3a is a valid ownership structure in Universe Types. The reference between object 4 and object 2 is stored in field `a` of class `C`. This field will be annotated with an `any` ownership modifier.

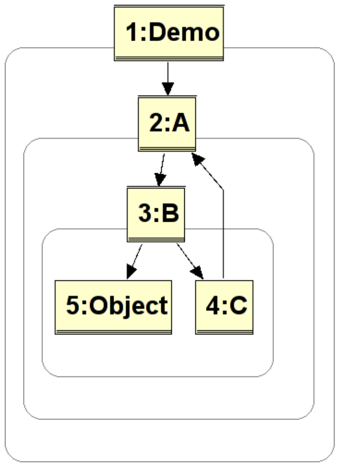
However, if `a.mod` is `true`, the non-pure method `off` is called on `a`. This results in a write reference from object 4 to object 2. In this case, the dominator graph does not represent a valid ownership structure because there is a write reference to an object in an enclosing context. This write reference can neither be typed with a `rep` nor with a `peer` modifier and is, therefore, not admissible in Universe Types. To solve this problem, we flatten the ownership structure to make the write reference from object 4 to object 2 admissible. This is done by raising the origin of the write reference (object 4) to the context that contains the destination of the write reference (object 2). This makes the two objects peers, and the write reference between them is admissible as 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. 3b. In this graph, all write references are from a direct dominator to an object it dominates or between objects with the same direct dominator. Therefore, this graph represents a valid ownership structure that can be expressed in Universe Types.

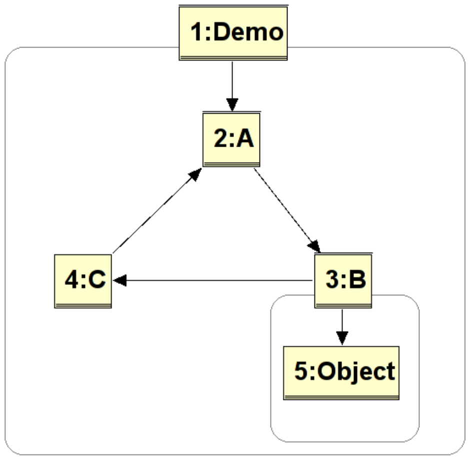
Our example shows that conflict resolution has to be applied repeatedly because resolving one conflict can cause others. Nevertheless, conflict resolution can be implemented efficiently without visiting the same write reference twice. To achieve that, we use a list of conflicting write references and process the list in a top-down way, that is, objects higher-up in the dominator graph are processed first. Moreover, we resolve conflicts that cross a large number of context boundaries before conflicts that cross fewer contexts. For details see [24].

2.4 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



(a) Dominator Tree



(b) After conflict resolution

Figure 3: Contexts are depicted by rounded rectangles. Owner objects sit atop the context of objects they own.

has to take into account both write and naming references in the EOG.

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

For example, imagine that method `testA` in class Demo is once called with `false` and once with `true` as the argument. Then we have two instances of class `A`, once with a deep ownership structure as in Fig. 3a and once with a flat structure as in Fig. 3b. The annotation for field `b` in class `A` is once `rep` and once `peer`. The algorithm then decides to use `peer` as annotation for field `b` and raises the non-conforming instance to a higher level. Because we raise an object together with all peers that reference it or are referenced by it, this step cannot create new conflicts in the ownership graph.

2.5 Output Universe Types

After the first four steps of the algorithm, we have determined ownership modifiers for field declarations, method parameters and results, and allocation expressions. The last step is to output these ownership modifiers and insert them into the source code, if it is available.

Local variables are not inferred from the EOG because that would require monitoring every assignment of a local variable, which would slow down the inference. As an implementation problem, Java JVMTI does not support monitoring of local variable assignments, and we deemed a solution using bytecode instrumentation too heavy-weight.

Inferring ownership modifiers for locals is very similar to Java's bytecode verification [23]. Both infer the types of local variables based on the types of fields and method signatures. Like bytecode verification, we symbolically execute the bytecode of a method body to obtain the ownership modifiers of local variables. This step might introduce downcasts when `any` references are used to modify objects. These casts are not guaranteed to succeed at runtime. Therefore, they should be reviewed by the programmer.

Fig. 4 shows the result of our inference for the example source code from Fig. 1. The ownership modifiers are inferred after processing program executions with and without command-line arguments. This source code complies to the Universe type system. By inserting the ownership modifiers into the source code, we ensure that future revisions of this code will maintain the ownership structure.

2.6 Static Methods

In Universe Types, static methods are either executed in the context in which the caller is executed or in the context owned by `this`. In the former case, the target type of the call to the static method has a `peer` modifier; in the latter case, it has a `rep` modifier. `any` is not permitted.

When we monitor the execution of a program, no object exists that corresponds to the target of the static call. In the EOG, we create an artificial target object as the receiver of a static method call. The relationship between the current object and the artificial object determines the ownership modifier for the static call. To enforce that the target of a static call does not have the `any` modifier, we always treat static method calls as non-pure. This creates a write reference in the graph and ensures that a `peer` or `rep` modifier is inferred.

Our treatment of static methods is illustrated by the example in Fig. 5. Consider the call x`.foo(`y`)`. The execution of `foo` affects three objects in the EOG: the receiver x, the parameter y, and an artificial target object for the call to `process`, say z. We add a write edge from x to z because x calls the static method. We also add a write reference from z to y because `process` calls a non-pure method on y. Since Universe Types do not allow `rep` modifiers in static methods, the latter write reference forces the parameter `p` of `process` to have a `peer` modifier. The modifier of the target type of the call to `process` is determined by the relation between the current receiver x and parameter y. Since `process` expects a `peer` parameter, y and the artificial target object z must have the same owner. Therefore, if x owns y, then x also owns z, and the annotated call will be `rep S.process(q)`. If x and y are peers, the call will be `peer S.process(q)`. In all other cases, step 2 of

```
public class Demo {
    public static void main(any any String[] args) {
        new peer Demo().testA(args.length > 0);
    }

    public void testA(boolean b) {
        new rep A(b);
    }
}

class A {
    boolean mod;
    peer B b;

    A(boolean m) {
        mod = m;
        b = new peer B(this);
    }

    void off() {
        mod = false;
    }
}

class B {
    peer C c;
    rep Object o;

    B(A a) {
        c = new peer C(a);
        o = new rep Object();
    }
}

class C {
    peer A a;

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

Figure 4: Running example with inferred ownership modifiers.

the inference will automatically adapt the relation between x and y and, thereby, the relation between x and z.

3. Implementation

Fig. 6 shows the architecture of the implementation. The tool is split into two parts: Sec. 3.1 describes the tracing agent, which monitors the execution of Java programs. Sec. 3.2 describes the inference tool, which determines the ownership modifiers.

3.1 Tracing Agent

We monitor a Java Virtual Machine (JVM) execution with a Java Virtual Machine Tooling Interface (JVMTI) agent written in C. JVMTI is the low-level interface provided by the Java Platform Debugger Architecture (JPDA) [20].

```
class S {
    static T process(T p) {
        p.nonpureOperation();
        return p;
    }

    T foo(T q) {
        return S.process(q);
    }
}
```

Figure 5: Example for static methods.

The agent receives events from the virtual machine and produces a trace file that documents the execution of the program. The trace file is in a simple XML format. Storing the execution of a program in a trace file gives the following advantages: (1) Multiple trace files can be generated to achieve good code coverage. In our example, one should trace the execution of class Demo once without any command-line arguments and once with an argument. (2) Interactive or long-running programs need to be traced only once for each desired code path. This trace file can then be reused later without requiring human interaction or recomputing results.

On the other hand, storing the trace files on disc and then parsing them again in the next phase sometimes leads to a performance overhead. In the beginning of this project, we investigated the Java Debug Interface (JDI) as high-level alternative to the low-level JVMTI. The JDI versions up to Java 5 did not provide enough information to allow our Universe inference, especially the value returned by a method was not accessible. In Java 6 the JDI API was enhanced and we investigate adding JDI support as an alternative source of program traces.

JVMTI does not provide the necessary events for array component updates. Therefore we used instrumentation of the Java bytecode to create artificial events for array updates.

3.2 Inference Tool

The main inference tool is an independent Java 5 application that performs the steps outlined in Sec. 2. It reads (multiple) trace files generated by the tracing agent and builds one Extended Object Graph from the available information. Then the dominators are determined, conflicts are resolved, multiple instances are harmonized, and the output is written to an XML file. The different steps of the algorithm are implemented as visitors that manipulate the EOG.

The application is configured by a simple XML file that determines what input and output files to use and which visitors and observers to use. This extensible architecture allows us to support a command line interface and the Eclipse plug-ins described in Sec. 4, and will also allow us to add JDI as an alternative input.

The output of our inference tool is an annotation XML file that contains the ownership modifiers for the encountered types. For this annotation XML file, we defined an XML schema that can provide ownership modifiers for the different types. If the source code of the traced program is available then the annotations can be inserted into the source code using a separate annotation tool we developed. Producing the output in XML will allow us to support several annotation tools, for instance, for the existing Universe syntax and for JSR 308-style annotations.

To build the correct EOG, we need to know which methods are pure. We use a separate annotation XML file as additional input to the inference tool to provide this purity information. This XML file has the same schema as the output file, which allows us to use the annotation editor, visualizer, and insertion tool to create

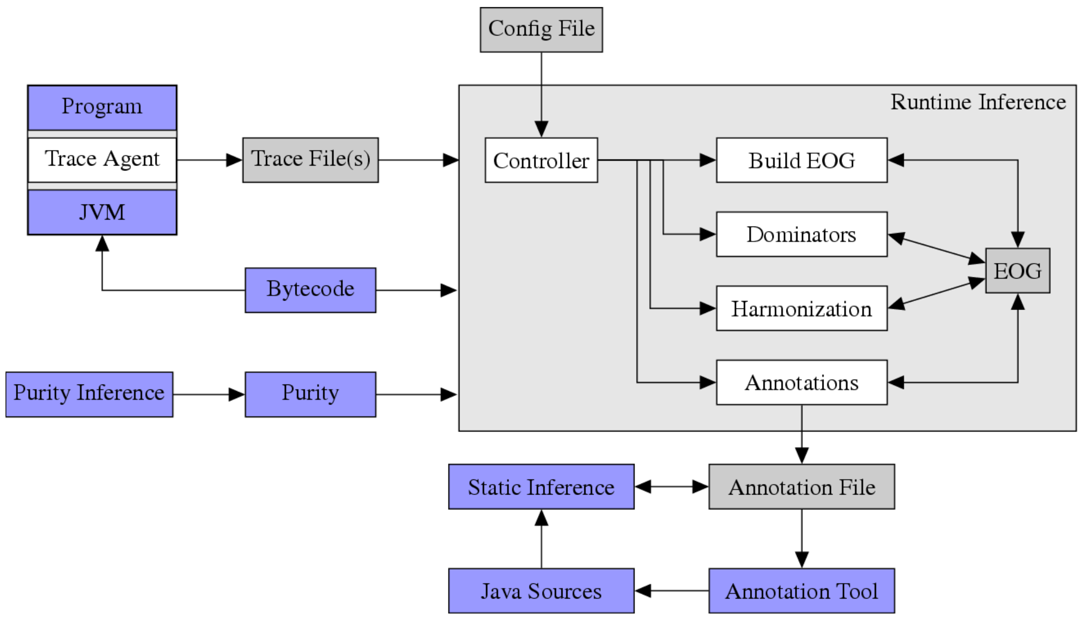


Figure 6: Architecture of the runtime inference tools. White boxes depict components of the inference tool. Boxes in light gray depict files and data structures that are part of the inference tool, and boxes in dark gray depict external components and files.

the input. To ease the creation of this purity information, we also implemented a purity inference tool [31, 17].

The XML file in Fig. 7 shows the result of applying the inference algorithm (without inference of local variables) to our running example (see Fig. 4 for the annotated source code). The Java structure is modeled in the XML structure, and the modifier attribute is used to provide the ownership modifier for the corresponding type or the purity for a method.

4. Eclipse Integration

To ease the usage of the command-line tools, we created a set of Eclipse 3.2 [15] plug-ins that integrate the runtime inference into the standard Java development environment.

4.1 Tracing

Eclipse allows one to execute Java programs directly from the IDE using “Run As” configurations. The programmer can use these configurations to set, for example, command-line arguments and the JVM to use. We added a new “Run As” configuration that allows one to trace program executions. The only additional information the user has to provide is the name of the trace file. The plug-in takes care of configuring the Java tracing agent correctly.

We provide the complete configuration information on a separate tab. This information can be copied into a script and allows the user to configure the tracing agent within Eclipse, but then use the command-line tool directly.

4.2 Inference

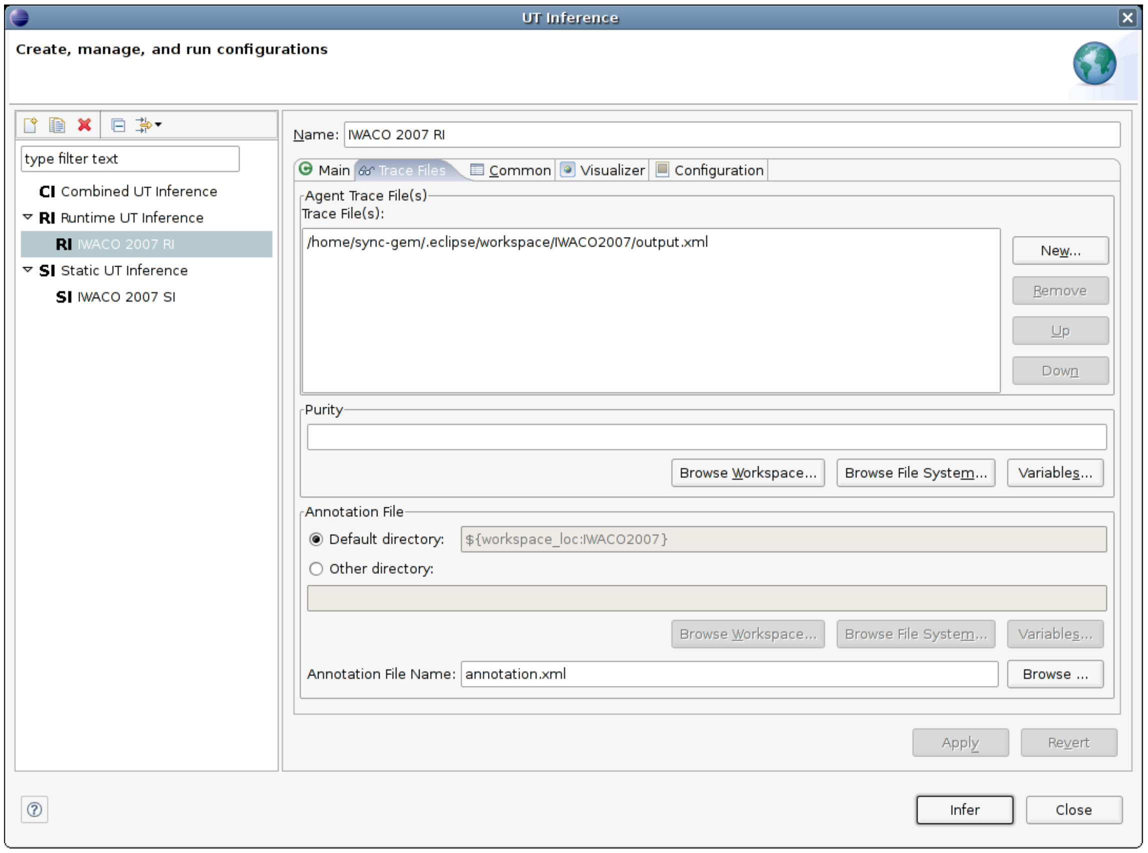
Once the program was traced, the Universe Types can be inferred with a separate plug-in. Similarly to the “Run As” dialog, we provide the possibility to manage different configurations. The main configuration tab (shown in Fig. 8) allows one to easily configure the trace files, purity information, and output file that should be used. Again, we provide a tab that allows one to use the configuration from the command line.

```
<?xml version="1.0" encoding="UTF-8"?>
<ann:annotations
  xmlns:ann="http://sct.inf.ethz.ch/annotations">
  <ann:head>
    <target>java</target>
    <style>types</style>
  </ann:head>
  <ann:class name="A">
    <ann:field modifier="rep" type="B" name="b"/>
  </ann:class>
  <ann:class name="B">
    <ann:field modifier="rep" type="C" name="c"/>
    <ann:field modifier="rep" type="java.lang.Object"
               name="o"/>
    <ann:method name="B" signature="A" modifier="">
      <ann:parameter index="0" modifier="any" type="A"
                     name="param0"/>
    </ann:method>
  </ann:class>
  <ann:class name="C">
    <ann:field modifier="any" type="A" name="a"/>
    <ann:method name="C" signature="A" modifier="">
      <ann:parameter index="0" modifier="any" type="A"
                     name="param0"/>
    </ann:method>
  </ann:class>
  <ann:class name="Demo"/>
</ann:annotations>
```

Figure 7: XML output of the inference tool.

4.3 Annotation Management

The result of the runtime inference is an XML file that contains the inferred ownership modifiers. This XML file can be either edited with the standard XML editor or with a special annotation editor. The annotation editor (shown in Fig. 9) allows one to edit the ownership information, for instance, by providing drop-down lists



Name: IWACO 2007 RI

Agent Trace File(s)
Trace File(s): /home/sync-gem/.eclipse/workspace/IWACO2007/output.xml

Purity:

Annotation File
[✓ Default directory:] ${workspace_loc:IWACO2007}
Other directory:

Annotation File Name: annotation.xml

Figure 8: Screen shot of the configuration dialog for the type inference. The user can set the tool options, for instance, which trace files to use and what output file to generate.

of possible ownership modifiers. If the source code of the program is available, we can automatically insert the ownership modifiers from the XML file into the source.

4.4 Visualizer

The flexible observer architecture that we chose for the inference tool allowed us to add a graphical visualizer to the inferer. This visualizer (shown in Fig. 10) uses the Eclipse Graphical Editing Framework (GEF) to display the extended object graph while it is built up and modified during the execution. This gives a clear understanding of how the program executes and how the inference algorithm works.

The visualizer adds a new toolbar to Eclipse. Here the user can set the zoom level, use automatic or manual layout of the graph, "play" the evolution of the inference algorithm, take a single step of the algorithm, or pause the animation. It further provides buttons that help in the manual layout of the graph. The automatic layout of the graph nodes is used by default. It automatically positions the nodes and routes the edges to have a nice diagram. It uses a simplex algorithm that tries to minimize the crossings of edges [16]. The manual layout can be used to manipulate the graph by hand.

The objects in the graph can be shown with and without the fields and methods that the corresponding class has. The display of this additional information follows the UML standard for object diagrams.

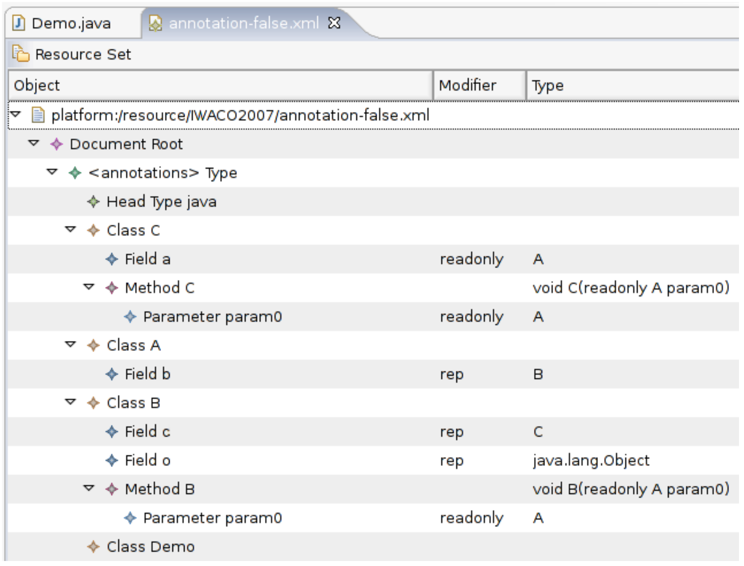
5. Conclusion and Future Work

This paper presented the current status of our work on runtime Universe type inference. We successfully used the tools in small case studies such as linked list and tree implementations. In these examples, the overhead of tracing the execution and the calculation of the ownership modifiers was reasonable. Even for small examples, the support for multiple trace files was very useful to increase the code coverage and, thus, the quality of the inferred ownership.

As future work, we plan to carry out non-trivial case studies. Inferring ownership for major applications will not only allow us to further evaluate and optimize our tools, but also provide insights into the structure of real applications. We expect this information to be valuable for further research on ownership in general.

Currently, our inference tool only works for non-generic Java. We recently developed Generic Universe Types [12, 11] and we will investigate whether runtime inference can be extended to generics. The problem is that genericity in Java 5 is implemented by erasure, that is, the type arguments are not visible to the virtual machine. It will also be interesting to study runtime inference in the presence of ownership transfer [28].

We plan to add JDI support to directly trace program executions without creating trace files. The inference visualizer is under active development and we have many ideas to make the interaction more convenient and to improve scalability to large object graphs. Examples include hierarchical folding of sub-trees, searching for instances of a particular class, and visual enhancements.



Object	Modifier	Type
platform:/resource/IWACO2007/annotation-false.xml		
Document Root		
<annotations> Type		
Head Type java		
Class C		
Field a	readonly	A
Method C		void C(readonly A param0)
Parameter param0	readonly	A
Class A		
Field b	rep	B
Class B		
Field c	rep	C
Field o	rep	java.lang.Object
Method B		void B(readonly A param0)
Parameter param0	readonly	A
Class Demo		

Figure 9: Screen shot of the annotation editor. The editor gives a tree view of the ownership information contained in an annotation XML file. Editing is simplified by drop-down lists of possible values.

Finally, we are integrating the runtime inference with our static inference tools [16]. This allows us to propagate and check the partial information that is inferred from program traces and ensures that the resulting annotations comply with the Universe type system.

Acknowledgments

This work builds on the Master's and semester theses of students at ETH Zurich: Frank Lyner [24] developed the first version of the command-line tool, Marco Bär [5] improved and extended the command-line tools, Marco Meyer [25] created the first annotation tool and visualizer, Paolo Bazzi [6] created the Eclipse plug-ins for runtime inference, David Graf [17] implemented the purity-inference tool, and Andreas Fürer [16] improved and created the Eclipse plug-ins and created the current version of the annotation tool and visualizer. Peter Müller's work was done at ETH Zurich.

References

[1] R. Agarwal and S. D. Stoller. Type Inference for Parameterized Race-Free Java. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 2937 of *Lecture Notes in Computer Science*, pages 149–160. Springer-Verlag, January 2004.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools, Second Edition*. Addison-Wesley, 2007.

[3] 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 *Lecture Notes in Computer Science*, pages 1–25. Springer-Verlag, 2004.

[4] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 311–330. ACM Press, 2002.

[5] M. Bär. *Practical Runtime Universe Type Inference*. Master's thesis, Department of Computer Science, ETH Zurich, 2006.

[6] P. Bazzi. *Integration of Universe Type System Tools into Eclipse*. Semester Project, Department of Computer Science, ETH Zurich, Summer 2006.

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

[8] 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.

[9] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 33(10) of *ACM SIGPLAN Notices*, 1998.

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

[11] W. Dietl, S. Drossopoulou, and P. Müller. Formalization of Generic Universe Types. Technical Report 532, ETH Zurich, 2006. `sct.inf.ethz.ch/publications`.

[12] W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In E. Ernst, editor, *European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science. Springer-Verlag, 2007. To appear.

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

[14] M. D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[15] The Eclipse Foundation. Eclipse — an open development platform. `www.eclipse.org/`.

[16] A. Fürer. *Combining Runtime and Static Universe Type Inference*. Master's thesis, Department of Computer Science, ETH Zurich, 2007.

[17] D. Graf. *Implementing Purity and Side Effect Analysis for Java Programs*. Semester Project, Department of Computer Science, ETH Zurich, Winter 2005/06.

[18] P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst. Dynamic inference of abstract types. In Lori L. Pollock and Mauro Pezzè, editors, *International Symposium on Software Testing and Analysis (ISSTA)*, pages 255–265. ACM, 2006.

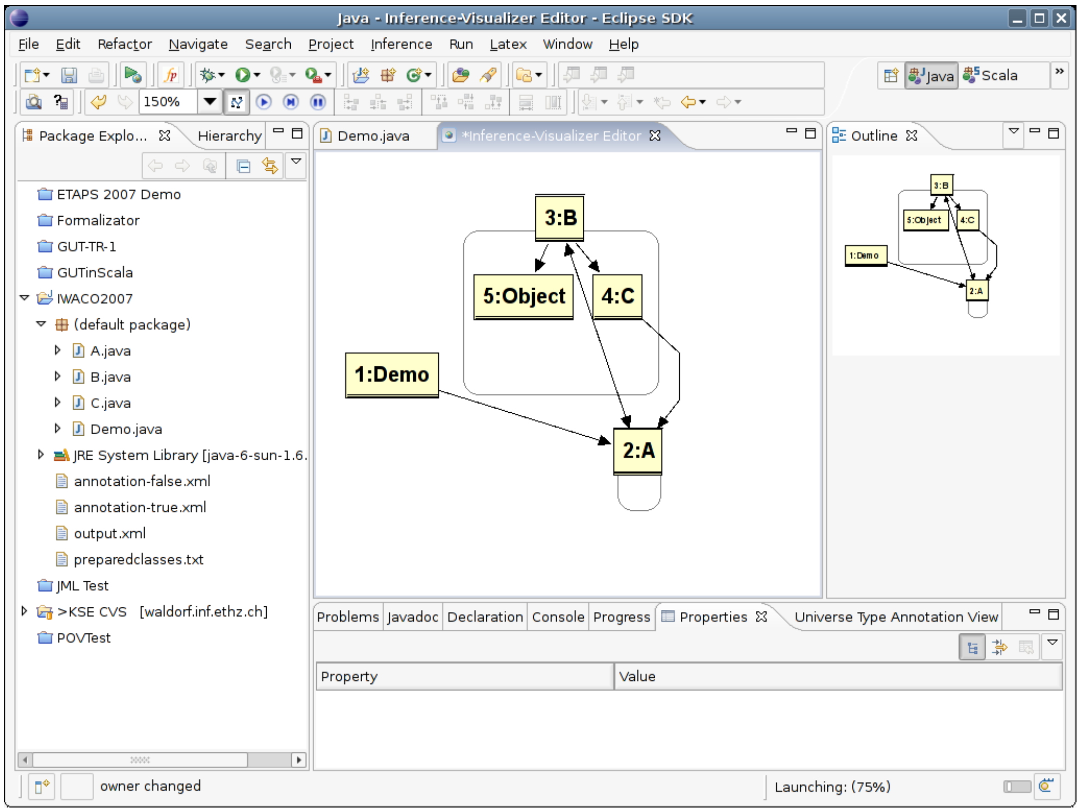

Java - Inference-Visualizer Editor - Eclipse SDK
File Edit Refactor Navigate Search Project Inference Run Latex Window Help
150%
Java
Scala
Package Explo...
Hierarchy
ETAPS 2007 Demo
Formalizator
GUT-TR-1
GUTinScala
IWACO2007
(default package)
A.java
B.java
C.java
Demo.java
JRE System Library [java-6-sun-1.6.
annotation-false.xml
annotation-true.xml
output.xml
preparedclasses.txt
JML Test
>KSE CVS [waldorf.inf.ethz.ch]
POVTest
Demo.java
*Inference-Visualizer Editor
3:B
5:Object
4:C
1:Demo
2:A
Outline
Problems Javadoc Declaration Console Progress Properties Universe Type Annotation View
Property
Value
owner changed
Launching: (75%)

Figure 10: Screen shot of the inference visualizer. The main editor in the center shows a (zoomable) image of the EOG. The outline view on the right helps in keeping the overview. The properties tab at the bottom gives additional information about selected elements. The status bar outputs information about the steps of the algorithm.

[19] S. E. Moelius III and A. L. Souter. An object ownership inference algorithm and its application. In M. T. Morazan, editor, *Mid-Atlantic Student Workshop on Programming Languages and Systems*, 2004.

[20] Sun Microsystems Inc. Java Platform Debugger Architecture (JPDA). `java.sun.com/javase/technologies/core/toolsapis/jpda/`.

[21] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, and J. Kiniry. JML reference manual. Department of Computer Science, Iowa State University. Available from `www.jmlspecs.org`, 2006.

[22] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, 1979. Available from `doi.acm.org/10.1145/357062.357071`.

[23] X. Leroy. Java bytecode verification: An overview. In *Computer Aided Verification (CAV)*, volume 2102, pages 265–285, 2001.

[24] F. Lyner. *Runtime Universe Type Inference*. Master's thesis, Department of Computer Science, ETH Zurich, 2005.

[25] M. Meyer. *Interaction with Ownership Graphs*. Semester Project, Department of Computer Science, ETH Zurich, Summer 2005.

[26] N. Mitchell. The runtime structure of object ownership. In Dave Thomas, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 4067 of *Lecture Notes in Computer Science*, pages 74–98. Springer-Verlag, 2006.

[27] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.

[28] P. Müller and A. Rudich. Ownership transfer in Universe Types. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2007. To appear.

[29] Matthias Niklaus. *Static Universe Type Inference using a SAT-Solver*. Master's thesis, Department of Computer Science, ETH Zurich, 2006.

[30] D. Rayside, L. Mendel, and D. Jackson. A dynamic analysis for revealing object ownership and sharing. In *Workshop on Dynamic systems analysis (WODA)*, pages 57–64. ACM Press, 2006.

[31] 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 *Lecture Notes in Computer Science*, pages 199–215. Springer-Verlag, 2005.

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

[33] H. Yan, D. Garlan, B. R. Schmerl, J. Aldrich, and R. Kazman. DiscoTect: A system for discovering architectures from running systems. In *International Conference on Software Engineering (ICSE)*, pages 470–479, 2004.