Exceptions in Ownership Type Systems

Werner Dietl and Peter Müller

ETH Zürich, Switzerland
{werner.dietl,peter.mueller}@inf.ethz.ch
http://www.sct.inf.ethz.ch/

Abstract. Ownership type systems are used to structure the object store into contexts and to restrict references between contexts. How to handle exceptions in these type systems has not been described in the literature. In this paper we analyze four viable designs for supporting exceptions in ownership type systems: (1) cloning exception objects during propagation; (2) using unique references to transfer exceptions between contexts during propagation; (3) treating exceptions as global data; (4) handling exceptions by read-only references that may cross context boundaries. We briefly describe our implementation of the fourth approach in the Universe type system.

1 Introduction

The basic idea of ownership models is to structure the object store hierarchically. Ownership models impose an acyclic *owner relation* on objects. Objects can have zero or one owner objects. All objects with the same owner object X are in one *context*, Γ, and X is called the owner of Γ. Objects without owner object are in the designated *root context*. Contexts form a hierarchy: Context Γ is a *child context* of context Δ if the owner of Γ is in Δ. We say that Δ is an *ancestor* of Γ if an object in Δ transitively owns Γ. Finally, we use *current context* to refer to the context in which the `this` object of the current method execution is. We do not treat static methods in this paper.

Whether a program follows the ownership model can be checked statically by ownership type systems. Several such type systems have been proposed for Java [1, 2, 4–6, 9, 10, 14–17]. Most of them allow only references from an object X to objects (1) in the same context as X, (2) owned by X, and (3) in ancestor contexts of the context in which X is. They guarantee the following *deep ownership* invariant [11] : Every chain of references from an object in the root context to an object X in another context passes through X's owner object. This invariant ensures that the internal representation of an object is not exposed to clients. In this paper, we focus on deep ownership, but will also consider a weaker invariant that allows read-only references to point to an object without going through its owner.

Most ownership type systems use ownership parameterization. In this paper, we use a simplified notation, where the keywords `peer`, `rep`, and `root` are used to denote references of the three kinds above. For example, an object of class `Person` in Figure 1 owns its `Car` object. `Car` objects refer to a global object representing the manufacturer's company. The tags in angle brackets will be replaced by different ownership annotations when we discuss different approaches to handling exceptions.

Conventional type systems without ownership treat exceptions as special return values that have a different control flow. They are propagated through the call stack until an exception handler is found or the program terminates abnormally. Besides so-called unchecked exceptions (`Errors` and `RuntimeExceptions`), exceptions that might be thrown by a method must be declared in the method's signature.

Exceptions cannot simply be treated as special return values in ownership type systems, because the exception might be handled in a different context than the context in which the exception was created. Consider the object structure of a `Person` object and the referenced `Car` and `Engine` objects in Fig. 2. Assume that method `Engine.start` creates an exception locally in the context Γ in which the `Engine` object is. (That is, we use `peer` for `<tag3>` and `<tag4>` in class `Engine`.) In this case, the exception cannot be propagated beyond the owning `Car` object without violating the ownership invariant.

```
class Person {
  rep Car mycar;

  void drive() {
    try {
      ...
      mycar.start();
      ...
    } catch( <tag1> CarException ce ) {
      System.err.println( ce.getOrigin() );
    }
  }
}
```

```
class Car {
  rep Engine engine;
  root Company manufacturer;

  void start() throws <tag2> CarException {
    ...
    engine.start();
    ...
  }
}
```

```
class Engine {
  void start() throws <tag3> CarException {
    ...
    throw new <tag4> CarException( this );
    ...
  }
}
```

```
class CarException extends Exception {
  peer Object origin;
  ...
}
```

Fig. 1. A `Person` object owns its `Car` object. Method `Person.drive` starts the person's car. A `CarException` is thrown if there is a problem. Such exceptions store a reference to the origin of the exception.

Technically, ownership type systems would require `<tag2>` to be `rep` because the exception is owned by the `Car` object. However, in this case method `Car.start` could only be invoked on `this`. Invocations on other receivers such as in the call `mycar.start` would be forbidden. Moreover, using `rep` and `peer` declarations in `throws` clauses is not viable for unchecked exceptions.

In this paper, we explain and analyze four approaches to supporting exceptions in ownership type systems:

1. Cloning exceptions from the context in which they are thrown to the context in which they are handled.
2. Transferring exceptions from the context in which they are thrown to the context in which they are handled.
3. Treating exceptions as global data that can be accessed from all contexts.
4. Propagating exceptions via read-only references, which may cross context boundaries.

2 Exceptions in Ownership Type Systems

In this section, we explain the four approaches to handling exceptions in ownership type systems and evaluate them w.r.t. the following criteria: (1) Expressiveness: what implementation patterns can be expressed by the approach? (2) Applicability: what specification overhead does the approach impose on programmers? Does it lead to performance overhead? (3) Implementation: how complex is the machinery in the type system that is needed to implement the approach?

2.1 Cloning

A poor man's version of supporting exceptions in ownership models is to clone the exception object and all objects (transitively) owned by the exception object (for instance, the message and stack trace strings) every time the exception is propagated across a context boundary. This approach

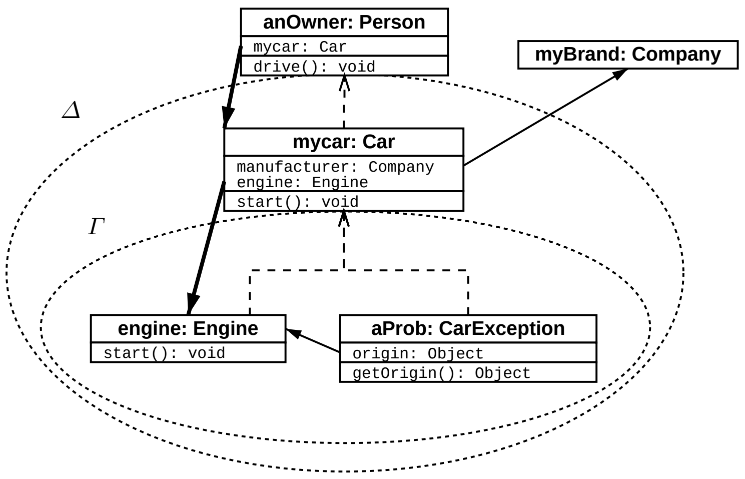

Fig. 2. Example object structure. The dashed lines represent the ownership relationship and point from an object to its owner object. Contexts are depicted by ellipses, and the owner of a context sits atop the ellipse.

guarantees that the exception is in the current context when it is handled. It can be used by any ownership type system.

With this approach, `tag1`–`tag4` in our example would be `peer` since cloning makes sure that the exception object is always in the current context. The example in Fig. 1 does not show explicit cloning of exceptions.

Expressiveness: Whereas cloning is expressive enough for most implementations with exceptions, it leads to problems in cases in which an implementation relies on the identity of the exception object, or when the exception object stores references to objects owned by other objects such as class `CarException` in Fig. 1. An exception of this class stores a `peer` reference to its origin, for instance, an `Engine` object. Type safety requires that the `Engine` object is cloned when the exception is cloned to an ancestor context. Otherwise, the clone in an ancestor context would have a reference to an `Engine` object that does not pass through the owning `Car` object. However, if the `Engine` object is copied, the identity of the origin is lost. Moreover, the cloning is not possible if the referenced objects are not cloneable as in our example, where `Engine` does not implement `Cloneable`.

Exceptions that store references to their origin can be found in several places in the Java API, for instance, in package `javax.swing.tree`. Objects of class `ExpandVetoException` store references to `TreeExpansionEvent` objects, which in turn reference the tree node that should be expanded. If this node is owned by any object, the problem described above occurs.

Applicability: Explicit cloning of exceptions, that is, by catching, cloning, and re-throwing, requires that all exceptions including unchecked exceptions would have to be caught after each call on a `rep` receiver. Even if all exceptions are caught in a single try-catch block around the whole method body, this leads to tremendous programming overhead. Alternatively, exceptions could be cloned implicitly when propagated across context boundaries. Implicit cloning, however, changes the language semantics. Moreover, cloning causes significant performance penalties, especially when an exception is propagated several times before it is caught.

Implementation: Supporting exceptions by cloning does not require major changes in existing ownership type systems. However, class `Throwable` must be declared cloneable and method `clone` must be overridden to perform a deep clone or a sensible clone [9]. That is, `clone` has to clone the receiver object and all objects it owns.

2.2 Transfer

Some ownership type systems such as SafeJava [4] support ownership transfer based on unique variables [11]. An exception mechanism based on ownership transfer is similar to cloning. Instead of creating new objects, an exception object and all objects it owns are transferred to ancestor contexts during propagation. Therefore, the exception is in the current context when it is handled. The advantage is that object identities are not lost during propagation and that there is no performance overhead.

In our example, this approach can again be expressed by choosing `peer` for `tag1`–`tag4` and adding uniqueness annotations. Fig. 1 does not show the declaration of unique variables and the transfers.

Expressiveness: Using transfer instead of cloning solves the problem of object identities, but does still not allow exceptions to store references to objects that are neither in the root context nor transitively owned by the exception. For instance, a `CarException` object cannot be transferred to an ancestor context, because the reference in `origin` would then break the ownership invariant.

Applicability: Transfer could be done implicitly, which avoids specification overhead.

Implementation: The main drawback of the transfer approach is the complexity of the underlying transfer technique [11, 7]. Transfer requires (externally) unique variables, which are not supported by most ownership type systems.

2.3 Global Exceptions

To avoid cloning or transferring exceptions and to make sure that all possible handlers of an exception have access to it, exceptions could be created in a designated context. For simplicity, we assume that this context is the root context. In this case, declaration and propagation of exceptions are straightforward. In our example, this approach can be expressed by choosing `root` for `tag1`–`tag4`.

Global exceptions can be used in ownership type systems that permit references to objects in ancestor contexts [1, 2, 4, 6, 9, 10, 17].

Expressiveness: Like the approaches based on cloning and ownership transfer, the global exceptions approach does not allow exceptions to reference objects other than global objects and objects owned by the exception.

Applicability: Global exceptions do not lead to specification overhead if `root` is chosen as default tag for `throws` and `catch` clauses, and for instantiation of `Throwable` objects. Global exceptions do not cause any runtime overhead.

As explained in the second author's thesis [14], references to objects in ancestor contexts are difficult to handle in program verification, in particular of class invariants because of the re-entrance (callback) problem. Assume that a method executed on receiver X temporarily violates X's invariant before it invokes a method on an object Y owned by X. If Y can have a reference to an object in an ancestor context, it might invoke a method on X, although X is not in a consistent state. This re-entrance problem is difficult to solve, but can be avoided by forbidding references to objects in ancestor contexts.

Implementation: Most ownership type systems support references to objects in a root context. For these type systems, the global exceptions approach does not introduce significant additional complexity.

2.4 Read-Only Exceptions

The Universe type system provides *read-only references* [14, 18, 8]. It guarantees a weaker ownership invariant, which allows read-only references to cross context boundaries. However, read-only references can only be used to read the state of the referenced object, but not to modify it. To check this limitation statically, only *pure* (that is, side-effect free) methods can be invoked on read-only references. Purity of methods has to be declared explicitly [13]. In the type system, the tag `readonly` is used to declare that a variable holds read-only references.

In the read-only exceptions approach, exceptions are created locally in the current context. That is, in class `Engine`, `tag4` is `peer`. Exceptions are then propagated as read-only references (`tag1`, `tag2`, and `tag3` are `readonly`). All objects in the program execution can have a read-only reference to the exception, which makes ownership transfer and cloning dispensable.

Expressiveness: This approach allows exceptions such as `CarException` to have (read-only or read-write) references to objects owned by other objects. Since the read-only protection is transitive (references gained through a read-only reference are again read-only), read-write references to an `Engine` are not leaked when a `CarException` is thrown.

Read-only exceptions have two limitations. First, read-only references lead to a weaker ownership invariant, since a reference chain from the root context to an object X that includes read-only references does not always pass through X's owner. This weaker invariant is uncritical for the verification of functional properties of sequential programs [14], but leads to problems for other applications of ownership type systems such as thread synchronization [5] or representation independence [3].

The second limitation is the case that an exception object is modified during propagation. To handle this case, one would have to clone the exception, make modifications to the clone, and throw the clone. Note that read-only references avoid the problems of cloning described in Subsection 2.1: Declaring `origin` in `CarException` to be `readonly` would enable cloning of exceptions without cloning all objects they refer to (sensible clone [9]). Since read-only references can cross context boundaries, the clone can have a read-only reference to the same `Engine` object as the exception thrown initially. Modifications of the `Engine` object have to be executed through the owners of the `Engine` object rather than directly through the exception.

Applicability: Similarly to global exceptions, the specification overhead can be reduced by using `readonly` as default tag for `throws` and `catch` clauses.

Implementation: Adding read-only references to an ownership type system does not increase the complexity of the type system significantly. Types for read-only references can be treated as supertypes of the types for the corresponding read-write references, thereby allowing assignments of references to objects in arbitrary contexts to variables for read-only references. Static checks are necessary for the purity annotations of methods and to ensure that read-only references cannot be used to modify the referenced objects. Moreover, the type system has to guarantee that the read-only property is transitive [14].

We implemented read-only exceptions as part of the Universe type system in MultiJava [12]. Since the Universe type system already supports read-only references, only the type checking for `throws` and `catch` clauses had to be changed. Moreover, we had to provide purity annotations for the methods in the Java libraries, which we took from JML specifications [13].

3 Conclusions

In summary, we think that both global exceptions and read-only exceptions are viable alternatives. However, the four presented approaches are not mutually exclusive. For instance, one could use global exceptions for unchecked exceptions such as `NullPointerExceptions`, but leave it to the programmer whether checked exceptions should be cloned, transferred, or propagated as read-only exceptions.

Due to its expressiveness, simplicity, and support for formal verification of functional correctness, we chose read-only exceptions for the Universe type system. This solution is easily integrated into our type system, in particular because it already provides read-only references. Read-only exceptions allow exceptions to refer to objects of encapsulated data structures and, in most cases, does not add any annotation or runtime overhead. In the unlikely case that an exception object is modified by the handler, the code has to be reorganized. In a nutshell, we think that read-only exceptions are a very practical way of supporting exceptions in ownership type systems.

Acknowledgments. We thank Gary Leavens for a discussion of the ideas presented in this paper. Moreover, we are grateful to the FTfJP reviewers for their helpful comments.

References

1. J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In *To appear in European Conference on Object-Oriented Programming*, 2004.
2. J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications*, 2002.
3. A. Banerjee and D. Naumann. Representation independence, confinement, and access control. In *Principles of Programming Languages (POPL)*, pages 166–177. ACM Press, 2002.
4. C. Boyapati. *SafeJava: A Unified Type System for Safe Programming.* Doctor of philosophy, Electrical Engineering and Computer Science, MIT, February 2004.
5. C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 211–230. ACM Press, 2002.
6. C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 213–223. ACM Press, 2003.
7. J. Boyland. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience*, 31(6):533–553, 2001.
8. J. Boyland, J. Noble, and W. Retert. Capabilities for aliasing: A generalisation of uniqueness and read-only. In J. Lindskov Knudsen, editor, *Object-Oriented Programming (ECOOP)*, number 2072 in Lecture Notes in Computer Science, pages 2–27. Springer-Verlag, 2001.
9. D. Clarke. *Object Ownership and Containment.* PhD thesis, University of New South Wales, 2001.
10. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 33(10) of *ACM SIGPLAN Notices*, October 1998.
11. D. G. Clarke and T. Wrigstad. External uniqueness is unique enough. In L. Cardelli, editor, *European Conference for Object-Oriented Programming (ECOOP)*, volume 2743 of *Lecture Notes in Computer Science*, pages 176–200. Springer-Verlag, 2003.
12. Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava: Design rationale, compiler implementation, and user experience. Technical Report 04-01, Iowa State University, Dept. of Computer Science, January 2004. Submitted for publication.
13. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06x, Iowa State University, Department of Computer Science, 2003. See `www.jmlspecs.org`.
14. P. Müller. *Modular Specification and Verification of Object-Oriented programs*, volume 2262 of *Lecture Notes in Computer Science.* Springer-Verlag, 2002.
15. P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.
16. J. Noble, R. Biddle, E. Tempero, A. Potanin, and D. Clarke. Towards a model of encapsulation. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, 2003. Available as technical report CS-TR-03-6 from `www.mcs.vuw.ac.nz/comp/Publications/archive/CS-TR-03/CS-TR-03-6.pdf`.
17. A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic ownership. Technical Report CS-TR-03-16, Victoria University of Wellington, 2003.
18. M. Skoglund. Sharing objects by read-only references. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology and Software Technology (AMAST)*, volume 2422 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.