Primitive Associations

Erik Ernst
University of Aarhus, Denmark
eernst@daimi.au.dk

Abstract

This position paper presents a very simple mechanism, *primitive associations*, and argues that this mechanism is worth careful consideration in connection with the kind of support for program correctness that grows out of mechanisms for ownership, controlled aliasing, sharing, escape analysis, and so on.

Categories and Subject Descriptors D.3 - PROGRAMMING LANGUAGES [*D.3.3 - Language Constructs and Features*]: Data types and structures

Keywords Ownership, confinement, alias control, primitive associations, inverse fields, path-restricted features.

1. Primitive Associations

Almost all object-oriented programming languages support a notion of references. A reference provides access to a specific object, and type systems are often mainly focused on specifying which (kinds of) objects are reachable from a given object. However, normally only little is known about the set of references referring *to* a given object—which we will designate as *incoming references*. Linear types [13], ownership types [9, 4, 3, 1, 11], escape analysis [10, 2], and other kinds of mechanisms and analyses help in establishing invariants or knowledge about these incoming references, and this may simplify reasoning about program correctness, especially because the sources of changes to objects and object graphs are simpler. However, we believe that it is useful to complement these techniques with a dynamic mechanism, namely *primitive associations*, because it is useful, simple, flexible, and understandable.

We define primitive associations to mean bidirectional references, i.e., a pair of references in two objects that refer to each other, see Fig. 1. Changes to these references must be restricted by the language semantics to enforce this invariant at all times: if A is an object and $A.\texttt{f}$ is a field in A that is part of a primitive association, then either $A.\texttt{f}$ is `NULL` or it refers to an object B such that B has a field $B.\texttt{g}$ which is the other half of that primitive association, and $B.\texttt{g}$ refers to the object A. Hence, the language must support statically decidable pairing of fields, and the run-time manipulation of fields which take part in a primitive association must occur atomically.

Given that the language semantics enforces this invariant, it is known for any given object A having a primitive association to another object B that no other object B' (respectively A') is in the same relation to A (resp. B). This may be interpreted as an ownership relation—that A owns B, or vice versa.

However, this ownership relation differs from more traditional ownerships by being more dynamic, because it may be changed by assignment. Other ownership related mechanisms would specify an owner via type declarations or type arguments and fix it at creation time for each owned object, thus disallowing the change of owner during the life-time of the owned object.

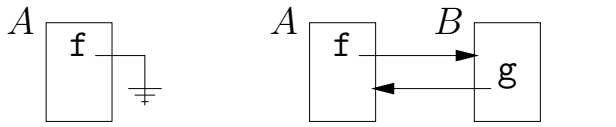

Figure 1. A primitive association is either `NULL` or cyclic

On the other hand, the dynamic flexibility of ownership by primitive associations provides fewer guaranteed properties at run-time. E.g., an inconsistency arises if the primitive association is modified during the execution of some operation which is only permitted for owners.

Primitive associations are closely related to *parent-child attributes* or *inverse fields* in JavaFX [12], because they also involve bidirectional references with language support for simultaneous updating. However, in this context we are interested in the ability to help managing uniqueness relationships rather than maintaining problem domain related constraints.

Note that it is easy to build associations of different arity than 1–1 based on primitive associations; for example, an array of length k may be used as an intermediate object to model a 1–k association.

2. Derived Correctness Properties

The main idea behind ownership is that it is easier to reason about the correctness of a program when ownership related invariants can be used to show that other invariants are maintained. The ownership related invariants are generally concerned with the exclusion of a (large) class of possible incoming pointers.

Consider for instance a `List` data structure which uses a number of `ListCell` objects to represent a linking structure and keep a reference to each of the contained objects. Now, invariants about the structure of each `List` object, including its `ListCells`, is easier to reason about if each `ListCell` is owned by one particular `List` object, and access to list cells is thereby restricted to come from the owner list or the list cells themselves. Conventional ownership mechanisms are well suited for this type of purpose; they associate each owned object (e.g., each `ListCell`) with an owner (a `List`) at creation time, and never change this binding during the lifetime of the owned object.

However, it is not always convenient to bind each owned object to one particular owner for its entire life-time. For example, it may be useful to move owned objects from one "owning context" to another. The main benefit of using primitive associations for ownership management is exactly this dynamic flexibility of being able to change owner during the lifetime of the owned object.

This property, however, creates challenges for exploiting ownership, i.e., to derive other correctness properties, because it gets harder to maintain a complex invariant that expresses a structural relation in the object graph of owned and owning objects when an assignment to a primitive association may suddenly change the owner. However, for the simple relationship that only involves the

two objects directly connected by a primitive association, there is a potential for reconciling these to opposing forces.

The concept required to express this is that of a *path-restricted feature*, i.e., a feature of an object that is only accessible via a specified path. Consider the pseudo-code example in Box 1 below:

```
class Person {
  private Wallet wlt <-> owner; // pr.ass.
  int pay(int value) {
    if (wlt.has(value)) {
      wlt.take(value); return value;
    } else {
      // error handling
    }
  }
}
class Wallet {
  private Person owner <-> wlt; // pr.ass.
  private int contents;
  public bool has(int value) {
    return (contents>=value);
  }
  restricted(wlt) void take(int value) {
    contents -= value;
  }
}
```

Box 1

In this example, the instances of the classes `Person` and `Wallet` are connected by a primitive association whose ends are named `wlt` and `owner`. In class `Wallet` there is a method `take` which is path-restricted by `wlt`. This means that an invocation of `take` is only allowed if it is on the form `wlt.take(...)` where `wlt` is the opposite end of a primitive association that connects a `Person` and this `Wallet`. The effect is that only the `owner` is allowed to call this method. Note that this differs from traditional ownership in that the person may choose to transfer the wallet to some other person.

3. Integrating Primitive Associations into gbeta

Primitive associations and the corresponding mechanism of path-restricted features are currently being implemented in the language gbeta [5, 8], where they complement a more traditional notion of ownership which is expressed using family polymorphism [6] and invisible mixins [7].

Family polymorphism includes a restricted form of dependent types: Classes are features of objects and thus two nested classes `Outer` and `Inner` give rise to a unbounded set of distinct types at runtime, because each instance of `Outer` contains its own, distinct class corresponding to the declaration named `Inner`. An invisible mixin is a mixin which is guaranteed to have a zero effect on the type of any class that it is added to—in other words, an invisible mixin can only add implementation, not interface. A consequence of this is that no code outside the mixin can refer to its declared features. Note that the notion of invisible mixins is in fact built on the notion of path restriction, because most of the characteristics of an invisible mixin are specified in terms of restrictions on paths.

Putting the two together, traditional ownership can be expressed by declaring owned classes in an invisible mixin. This is now complemented with the ability for owned object structures to include temporary ownership based on primitive associations and path restrictions.

It is our impression so far that this combination of life-time ownership and temporary ownership makes it easier to express practical program designs and still have a better basis for reasoning about the possible run-time object structures than that which is offered through traditional ownership or traditional unrestricted (un-owned) objects.

4. Conclusion

This position paper presented some preliminary thoughts about the usefulness of the very simple construct of primitive associations (aka inverse fields), used to express a dynamic kind of ownership. The notion of path-restricted features was created as a consequence of this analysis, as a special case of earlier work on so-called invisible mixins. We believe that this combination of mechanisms provides a simple and useful complement to traditional ownership mechanisms.

Acknowledgments

The IWACO reviewers provided some very helpful comments on this work.

References

[1] Jonathan Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanism. In Martin Odersky, editor, *Proceedings ECOOP*, volume 3086 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2004. ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings.

[2] Bruno Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In *Proceedings POPL '98*, pages 25–37. ACM SIGACT and SIGPLAN, ACM Press, 1998.

[3] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications (OOPSLA-02)*, volume 37, 11 of *ACM SIGPLAN Notices*, pages 292–310, New York, November 4–8 2002. ACM Press.

[4] David Clarke. *Object Ownership and Containment*. PhD thesis, School of Computer Science and Engineering; University; of New South Wales, Australia, July 12 2001.

[5] Erik Ernst. *gbeta – A Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, DEVISE, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 1999.

[6] Erik Ernst. Family polymorphism. In Jørgen Lindskov Knudsen, editor, *Proceedings ECOOP'01*, LNCS 2072, pages 303–326, Heidelberg, Germany, 2001. Springer-Verlag.

[7] Erik Ernst. Reconciling virtual classes with genericity. In *Proceedings JMLC'06*, LNCS 4228, pages 57–72, Oxford, UK, September 2006. Springer-Verlag.

[8] Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *Proceedings POPL'06*, pages 270–282, Charleston, SC, USA, 2006. ACM.

[9] James Noble, John Potter, David Holmes, and Jan Vitek. Flexible alias protection. In *Proceedings of ECOOP'98*, Brussels, Belgium, July 20 - 24 1998.

[10] Young Gil Park and Benjamin Goldberg. Escape analysis on lists. In *Proceedings of the 5th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 116–127, 1992.

[11] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic ownership for generic java. In Peri L. Tarr and William R. Cook, editors, *Proceedings OOPSLA*, pages 311–324. ACM, 2006.

[12] Inc. Sun Microsystems. Javafx script – an overview. `http://www.sun.com/software/javafx/script/`, July 2007.

[13] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*, Sea of Galilee, Israel, April 1990. North Holland. IFIP TC 2 Working Conference.