On Ownership and Accessibility

Yi Lu and John Potter

Programming Languages and Compilers Group
School of Computer Science and Engineering
University of New South Wales, Sydney
{ylu, potter}@cse.unsw.edu.au

Abstract. Ownership types support information hiding by providing statically enforceable object encapsulation based on an ownership tree. However ownership type systems impose fixed ownership and an inflexible access policy. This paper proposes a novel type system which generalizes ownership types by separating object accessibility and reference capability. With the ability to hide owners, it provides a more flexible and useful model of object ownership.

1 Introduction

The object-oriented community is paying increasing attention to techniques for object level encapsulation and alias protection. Formal techniques for modular verification of programs at the level of objects are being developed hand in hand with type systems and static analysis techniques for restricting the structure of runtime object graphs. Ownership type systems have provided a sound basis for such structural restrictions by being able to statically represent an extensible object ownership hierarchy. The trick to ownership systems is to hide knowledge of the identity of an object outside its owner. This form of information hiding is useful for modular reasoning, data abstraction and confidentiality.

Ownership types support instance-level information hiding by providing a statically enforceable object encapsulation model based on an ownership tree. Traditional class-level private fields are not enough to hide object instances. For example, an object in a private field can be easily returned through a method call. However, the encapsulation mechanism used by ownership types is still not flexible enough to express some common design patterns such as iterators and callback objects. Moreover, ownership types, to date, lack ownership variance. This means, for instance, that all elements stored in a list must be owned by the same owner due to the recursive structure of the list.

This paper proposes a novel type system which generalizes ownership types with an access control system in which *object accessibility* and *reference capability* are treated orthogonally. The rationale behind this mechanism is that one only needs to hold access permission for an object in order to use it; the capability of the object can be adapted to the current access context. This allows more flexible and expressive programming with ownership types.

Our system allows programmers to trade off flexibility/accessibility with useability/capability. We allow object accessibility to be variant; intuitively it is

D. Thomas (Ed.): ECOOP 2006, LNCS 4067, pp. 99–123, 2006.

safe to allow accessibility to be reduced as computation proceeds. On the other hand, we allow reference capability associated with an object to be abstracted in contexts where it is used. Our resulting type system is flexible enough to encode iterator and callback-like design patterns. It also allows objects, such as recursive data structures, to hold references to elements owned by different objects.

This paper is organized as follows: Section 2 gives an introduction to object encapsulation and the mechanisms used in ownership types; it also discusses the limitations of ownership types. Section 3 proposes the variant ownership object model and its key mechanisms with some program examples. Section 4 presents a small object-oriented programming language to allow us to formalize the static semantics, dynamic semantics and some important properties. Section 5 follows with discussion and related work. Section 6 briefly concludes the paper.

2 Ownership Types

Earlier object encapsulation systems, such as *Islands* [13] and *Balloons* [2], use full encapsulation techniques to forbid both incoming and outgoing references to an object's representation. However, full encapsulation techniques are overly strong, because outgoing references from the representation are harmless and are often needed to express typical object-oriented idioms. Ownership types [11, 10, 7] provide a more flexible mechanism than previous systems; they weaken the restriction of full encapsulation by allowing outgoing references while still preventing representation exposure from outside of the encapsulation. The work on ownership types emanated from some general principles for *Flexible Alias Protection* [20] and the use of dominator trees in structuring object graphs [22].

Ownership type systems establish a fixed per object ownership tree, and enforce a reference containment invariant, so that objects cannot be referenced from outside of their owner — an object owns its representation and any other external object wanting to access the representation must do so via the owner. Ownership types are parameterized by names of runtime objects, called *context*s in the type system [10]. Contexts include all objects and a pre-defined context called world which is used to name the root of the ownership tree. The world context is in scope throughout the program. All root objects are created in the world context and all live objects are reachable from the root objects.

In defining ownership the first context parameter of a class is the owner of the object. An object's owner is fixed for its lifetime, thus naturally establishing an *ownership tree* in the heap. The rest of the context parameters are optional; they are used to type and reference the objects outside the current encapsulation, which is how ownership types free objects from being fully encapsulated, as in early approaches to alias protection.

Types are formed by binding the formal context parameters of a class. In order to declare a type for a reference, one must be able to name the owner that encapsulates the object. An encapsulation is protected from incoming references because the owners of objects inside the encapsulation cannot be named from the outside. The key mechanism is the use of variable this to name the current