Capabilities for Sharing*
A Generalisation of Uniqueness and Read-Only

John Boyland¹, James Noble², and William Retert¹

¹ University of Wisconsin-Milwaukee, USA, {boyland,williamr}@cs.uwm.edu
² Victoria University of Wellington, New Zealand, kjj@mcs.vuw.ac.nz

Abstract. Many languages and language extensions include annotations on pointer variables such as “read-only,” “unique,” and “borrowed”; many more annotations have been proposed but not implemented. Unfortunately, all these annotations are described individually and formalised independently – assuming they are formalised at all. In this paper, we show how these annotations can be subsumed into a general capability system for pointers. This system separates mechanism (defining the semantics of sharing and exclusion) from policy (defining the invariants that are intended to be preserved). The capability system has a well-defined semantics which can be used as a reference for the correctness of various extended type systems using annotations. Furthermore, it supports research in new less-restrictive type systems that permit a wider range of idioms to be statically checked.

1 Introduction

Pointers to objects in imperative languages provide a powerful programming tool but one that has the potential to create hard-to-spot errors [23]. Even ignoring issues such as object allocation and deallocation, aliasing – sharing mutable state – can lead to undesired situations where two parts of a program unintentionally communicate.

Problems of representation exposure are well known [27]. For example, manipulating reference fields of a link object supposedly encapsulated within a linked list can cause the linked list to return incorrect values or loop indefinitely. Argument exposure is a more subtle problem: if an element of a collection such as a binary search tree, sorted list, or hash table is modifiable from outside, changing the element can break the invariant of the collection structure

* Work supported in part by the National Science Foundation (CCR-9984681) and the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF under contract F30602-99-2-0522. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the National Science Foundation, Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.
of which it is a part. For a recent summary of these issues, with particular attention to object-oriented systems, see the summaries from the IWAOOS 1999 workshop.

### 1.1 Annotations for Controlling References

To enable programmers to manage aliasing in their programs, and to support optimisation, many programming language mechanisms have been provided (and many more proposed) that restrict the use of pointers and objects in programs. For instance, many languages now support a *read-only* annotation (such as C++’s `const` and Java’s `final`) which prevents references from being changed. Many more flexible annotations have been proposed. If an object is declared to be completely *immutable*, that is, its state can never be changed via any reference, sharing that object always is benign. Alternatively, if a reference to an object is *unique*, that is, it is the only reference to that object, then that object can always be mutated without affecting other parts of the program.

Figure 1 summarises a number of the most important annotations that have been developed to date.

The proliferation of annotations causes a number of problems. Each annotation is typically described independently, without reference to other annotations. Similarly, each annotation is formalised independently, in a formal system suited to that particular purpose. The names given to annotations are quite arbitrary – for example, we believe that Hogg’s `read` in Modula-3, JAC and Universes, and some senses of C++’s `const` all have the roughly the same semantics that we call *read-only*. Subtle differences remain because each annotation is defined within a different context. For instance, *read* (reference) variables in Hogg’s Island system may not appear on either side of an assignment. Annotations’ semantics can be as arbitrary as their names, of course, so a read-only annotation could protect a variable, a single object (as in the example above), or provide transitive protection (as in Flexible Alias Protection’s `clean` or JAC’s `readonly`). The same annotation may have different meanings in a single system depending where it is applied: “unique” in Hogg’s