Implementing Relationships using Affinity
(Position Paper)

Stephen Nelson
Victoria University of Wellington
stephen@ecs.vuw.ac.nz

David J. Pearce
Victoria University of Wellington
djp@ecs.vuw.ac.nz

James Noble
Victoria University of Wellington
kjx@ecs.vuw.ac.nz

ABSTRACT

OO languages typically provide one form of object equality, known as reference equality, whereas two objects are equal only if they are the same object; two objects which are structurally identical are not considered equal. Thus, programmers who require a more refined notion of equality must define their own operator. Programmer-implemented equality operators tend to rely on informal notions of partial and temporal object immutability which can be prone to error. This is a particular problem for objects used in collections which depend on equality. This paper discusses Affinity: an untyped, object-oriented language with a powerful equality operator based on EGAL [2] and support for object-keying and immutability. Affinity is designed to provide coherent and elegant support for object equality, reducing programmer burden and error potential.

1. INTRODUCTION

Language-defined equality operators are a common and important language feature. Functional languages often provide several equality operators including structural equalities, whilst Object-Oriented languages typically only provide reference-equality operators. This forces OO programmers to implement their own equality operators, which is error prone. Programmer-defined equality methods can also prevent languages from making optimisations based on known properties of equality implementations.

Java provides two equality operators for objects: a primitive reference comparison (==) and the (.equals(...)) method which defaults to reference equality, but can be overridden. Programmer-implemented notions of equality are a common source of error [7, 6]. Nelson et al. [8] observe that in most cases programmers do not overwrite the default equality, and when they do the equality usually depends on immutable state.

Unusual or incorrect equality implementations are particularly problematic for collections and relationships. The Java collections API [1] imposes non-trivial constraints on object equality, and relationship systems often avoid equality concerns by using reference equality [5, 10]. Vaziri et al. proposed a system for automatically defining equality where possible [13]. Their system allows special types called Relation Types to nominated fields as key fields. They then generate an equality method for the class which uses these key fields to determine object equality, performing a structural equality up to mutability.

This paper builds on ideas from Vaziri et al.’s work [13]. We introduce Affinity, an untyped language which provides a primitive equality operator similar to EGAL, an operator proposed by Baker [2]. This operator can be used for identity equality and for more complex structural equalities on immutable state. Affinity also includes several other features which encourage good practices relating to object equality. These include immutability, late initialization of immutable and partially immutable objects, and encapsulation for objects with dependent equality. Finally, we introduce a novel approach to inheritance through delegation, which, combined with our approach to equality and immutability, provides a unified model for objects and roles. We believe that Affinity allows the implementation of collections and relationships in a safer, more intuitive manner than by relying on reference or programmer defined equality.

2. AFFINITY OVERVIEW

Affinity is a dynamic, untyped object-based language which is motivated by the design decision to use a single equality operator which performs equality up to mutability (Egal), instead of providing an operator for comparing reference. To support this decision, Affinity introduces four novel language features.

1. Keys. Egal does not support structural comparisons of mutable objects, so Affinity supports automatic unification of mutable objects with identical immutable state. This allows programmers to ensure that objects which are structurally identical on an immutable subset of their state are also egal.

2. Spaces. Previous work using automatic unification of objects or object lookup based on key fields (e.g. [13, 5, 10]) identified that this can cause problems for garbage collection and encapsulation (discussed in §2.3). Affinity introduces a feature called Spaces to encapsulate keyed objects.

3. Immutability. Affinity provides a clone operation for mutable objects which returns an immutable snapshot of that object. This supports late initialisation.

4. Inheritance. Affinity provides delegation-based single inheritance. It also supports multiple objects delegating to a single parent object, thus enabling roles [3]. Affinity-style delegation is designed to work intuitively with egal.

In the remainder of the paper, we will discuss each of these four design decisions in detail. Our motivation for these design decisions has been strongly influenced by experimental work exam-
ing how objects behave in practice [8]. In particular, we have observed the following types of object equality:

- **Reference Equality.** The programmer expects objects to be equal if they are references to the same object.
- **Value Equality.** Objects are entirely immutable, and the programmer expects the objects to be equal iff all of their fields are equal, including those objects reachable via fields.
- **Keyed Equality.** Objects have some immutable fields, and the programmer expects objects to be equal iff their immutable fields are equal.
- **Post-initialiser Value Equality.** Objects become immutable. That is, once the object has been fully initialised (which may not happen until after construction), the programmer expects objects to be equal iff their fields are equal.

The final type, post-initialiser value equality, can also occur as a form of keyed equality, but this is symmetric to the value/keyed equality cases so we do not discuss it separately. In [8] we also discussed a fifth type of equality: where the programmer changes equality after object creation. However, we did not observe this type of equality in practice so we do not consider it here.

### 2.1 Egal

Affinity does not provide a reference equality operator. Instead, there is a single operator, $\equiv$, called egal [2]. Egal performs structural equality up to mutability; that is, objects with mutable fields are compared by reference, whereas objects without mutable fields (immutable objects) are compared by recursively comparing their fields. An immutable object is never egal to a mutable object, and egal is an equivalence relation. Unlike a programmer defined equality method like Java’s `equals()`, egal is a stable equality; it depends only on immutable properties of objects so it cannot change over time.

Affinity indicates that fields are immutable using a `final` modifier:

```java
class A { final a; final b; }
class B { final a; var b; }
```

Egal will use a recursive comparison for class A because all of its fields are immutable and reference equality for class B, because it has a mutable field `b` which could change, changing the equality of instance of B.

Egal is more restrictive than comparison operators like Java’s `equals()` which depend on mutable state, but is more flexible than comparison operators like Java’s `==` because it traverses object graphs recursively (upto mutability).

Egal provides support for the first two types of equality that we identified: reference equality for mutable objects, and value equality for immutable objects (where all of the fields of the object are immutable).

### 2.2 Keys

Egal does not support structural comparisons of mutable objects, so Affinity provides a mechanism called Keys for unifying mutable objects which are structurally equivalent on an immutable subset of their state. Mutable objects which are keyed will be egal if the immutable subset of their state is structurally equivalent, even though egal will use reference comparisons to compare them.

Programmers can use keys to associate mutable state with objects while still appearing to have equality other than reference equality: the language still uses references, but ensures that two objects are egal iff they are the same object (i.e. reference equality). This supports the third type of equality we identify: keyed equality. Keyed objects are very similar to Vaziri et al.’s Relation Types [13].

### 2.3 Spaces

Keying can result in collisions when an object is created which has the same immutable (key) fields as another object of the same class. There are various ways of dealing with these collisions: the language could throw an exception if a duplicate object is created, or it can return the previously created object. Throwing an exception can introduce bugs that are very hard to diagnose: if keyed objects with the same immutable fields are created in different parts of the program, by different threads, then it can be hard to detect where the other object is created. The second solution, returning existing objects, breaks encapsulation by allowing access to objects created externally. In addition, it can causes problems for garbage collection. Usually, a keyed object can be collected if one or more of its keys is collected, and there are no references to it. If an object is keyed entirely on primitive state (e.g. integers) then the object cannot be collected as its keys will never be collected.

To reduce the chance of collisions and to help resolve them we propose key-spaces, a concept similar to Vaziri et al.’s Scopes. Key spaces are objects that are used as additional keys without being explicitly added to the object as state. This means that if the space is collected then the object cannot be recreated, so it may be collected when there are no more references to it.

Unlike Scopes, a space parameter can be any mutable object:

```java
class Foo { final j; final k; var l; }
class Bar { final j; final k; var l; }
```

The objects `a` and `b` are retrieved from the key spaces of the objects `x` and `y` respectively. This is equivalent to adding an additional key to the `Foo` class, but indicates to the language that these parameters are spaces, which are not tied to the lifetime of the object. Only mutable objects can be used as spaces, which ensures that every object has at least one mutable key.

The uniqueness guaranteed by keying applies only to other objects in the same key space. Because `a` and `b` are in different spaces they do not collide, so they are distinct and may have different state. Objects `b` and `c` are in the same space, so they do collide. Updating `b.l` will also change `c.l` as they are unified to the same object.

Objects which are not given a key space are not keyed. This encourages the use of key spaces to provide encapsulation and modularity.

### 2.4 Mutability

Objects are always considered mutable until their constructor has finished. Until this happens, calling egal on the object returned by `this` will use reference comparison. Once the constructor has completed, the language will designate the object as immutable if it has no mutable fields, or keyed, if they were given a space parameter. A programmer can create an immutable version of a mutable object at any point in the program by coercing it using the `value()` method on the object. The immutable version of the object will not be egal to the object it was created from, and if the original object was keyed then the immutable version will not be.

Coercing from mutable to immutable should be implemented efficiently, and has several useful consequences:

- It allows mutable objects to be compared on all of their fields.
Affinity supports two patterns for using inheritance: class based inheritance and role based inheritance. The distinction between these is that class inheritance reuses existing code by creating a new object to delegate to, and role inheritance reuses existing objects by using an object which has already been created. Both types of inheritance provide polymorphism.

The example in the previous section uses class inheritance. Foo objects create new instances of Bar objects which they delegate to. The Bar objects do not exist before the constructor runs, it is entirely up to the implementation of the Foo constructor whether or not they can escape Foo object.

This type of inheritance is called class inheritance because it allows the reuse of existing classes without exposing new instances of the delegate class, and is similar semantically to inheritance in traditional class-based languages.

Role inheritance occurs when an object delegates to an object that already exists. If a constructor takes a parameter which it assigns to super then the object will delegate to that other object, but as there may be other references to the delegate object the new object cannot make assumptions about the delegate object’s state.

class Baz {
    Baz(a) {
        super; a
    }
}

foo: new Foo(1, 2)
foo.c: 3
baz: new Baz(foo)
foo.c: 4
print baz.c; /* prints '4' */

The object baz uses role based delegation. It doesn’t have any control of the value of foo.c, which can change without any messages being passed to baz.
Rather than preventing duplicates in the global scope of the program, we are ensuring that they don’t occur in a particular context, in this case our university.

We can define courses similarly:

```java
class Course {
    final name;
    var time
}
```

Courses are keyed on their name, so only one course with each name can exist in a given key-space.

To store information about the courses a student is enrolled in, and the marks they receive we can create a relationship between students and courses:

```java
class Attends {
    final student;
    final course;
    var mark;
}
```

Relationships do not need special syntax in Affinity. They are declared as classes, and key spaces can be used to enable multiple instances of the same relationship. By keying on the student and course the language will ensure that there is only one link in a given space for a particular student and course - the student can only enrol in this course once in a particular relationship (space).

We can now create some students and courses, and enrol the students in courses:

```java
uni = "Victoria University"
prog = Course("programming", 4)
types = Course("Type Systems", 4)
alice = Student("Alice", 1001)
bob = Student("Bob", 1002)
Attends[uni](alice, prog)
Attends[uni](alice, types)
Attends[uni](bob, prog)
```

The relationship is also defined as a class of objects, which means that it cannot be accessed from the participants directly. However, the programmer can recreate the object (taking advantage of key space collisions) to retrieve it:

```java
Attends[uni](alice, prog).mark('A')
```

Alternately, the programmer can use key space accessors. Affinity builds internal tables to keep track of keyed parameters, and key spaces provide an interface for the programmer to query these tables. Access to the key space for a particular key is provided using the same square braces as used by the key parameter:

```java
print [uni].Attends(alice, prog).mark()
```

// prints 'A'

Key space queries can also use wildcards to retrieve sets of objects, which can then be traversed:

```java
for (a in [uni].Attends(alice, *)) {
    print a.student.name " got " a.mark " for " a.course.name
}
```

// prints:
// Alice received A for Programming
// Alice received undefined for Type Systems

Finally, suppose we want to allow some students to teach courses. We can create a new class called tutor for these students, and use role-based inheritance to link them to the students:

```java
class Tutor {
    final course;
    Tutor(student, course) {
        super: student;
        this().course: course
    }
}
```

This allows us to take existing students and create (or retrieve) tutor roles for them teaching particular courses.

4. CONCLUSION

Affinity’s combination of egal, keys, spaces, immutability and inheritance provides excellent support for object equality, addressing each of the object equality situations we considered without requiring programmer-defined equality. It also provides elegant and succinct support for relationships using existing language tools.

There are several open questions remaining. In particular, this paper does not consider space inheritance, which is related to relationship inheritance. We plan to extend Affinity to address this.

This paper builds on several important pieces of related work. In particular, Vaziri et al.’s Relation Types [13], but also various other relationship systems [4, 5, 10, 9, 11].

We are not aware of any mainstream languages that have adopted egal, which is a pity as it is a particularly nice operator to use. Vaziri et al.’s Relation Types use an equality operator very similar to our egal, their work has strongly motivated ours. We hope the extensions we have made, such as spaces, immutability casts, our inheritance mechanisms, and the applications to roles and relationships, are in the spirit of the authors work.

5. REFERENCES

[2] H. G. Baker. Equal rights for functional objects or, the more things change, the more they are the same. SIGPLAN OOPSLA, 1993.