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, where 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 are 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 initialisation 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-

RAOOL '09, July 7 2009, Genova, Italy

ining 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 *iff* 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, =, 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:

```
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 egality 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 identifier: 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 being 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:

```
class Foo { final j; final k;  var l; }
a: Foo[x](1,1)
b: Foo[y](1,1)

c: Foo[y](1,1)
```

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.

- It allows keyed objects from different spaces to be compared using egal.
- It allows programmers to use post-initialisers on otherwise immutable objects.

Immutability coercion supports the fourth type of equality we identified: Post-initializer value equality. Programmers can create objects, modify them in post-initializers, then coerce them into immutable objects which can be compared with egal.

2.5 Inheritance

Affinity uses delegation to pass messages between objects in an inheritance hierarchy, which is similar to object languages like Self [12]. Affinity supports two forms of inheritance: class-based and role-based, both using delegation.

Objects in affinity may have a single super object which they delegate to if they receive messages they don't understand. The super object may be provided to the constructor, or created within the constructor by the programmer. Unlike most languages which provide **this** and **super** as fields, Affinity provides a single method, **this**() to access the current object. **this**() will always return the object that was the target of the current method, even if the method is defined on a super object. This prevents the programmer obtaining a reference to the super object, which is important because an object and its super object are not egal, unlike Java. Super objects are implicit keys (immutable fields).

2.5.1 Constructors

Objects are constructed (or retrieved, in the case of keyed objects) using implicit or explicit constructors which may define the object's immutable state or call other constructors. Affinity provides implicit constructors which initialise all final fields from parameters, but programmers may define their own constructors also.

```
class Foo {
    final a; final b; var c;
    Foo(a, x) {
        this().a: a;
        this().b: Bar();
        super: Bar();
}   }
```

In this example instances of `Foo` can be constructed using the implicit constructor (e.g. line 5, `Foo(a, null)`) , or by calling the explicit constructor:

```
Foo(a)
```

If the programmer wishes to use inheritance for the object then they must use an explicit constructor. Inheritance is defined in the constructor by setting the special **super** field, as shown in the example. This can be done at any point in the constructor. Constructors cannot modify or read from mutable state other than their local variables and they cannot call methods other than constructors or **this**()[1]. It would be an error to set or read from `c`, for example.

The restrictions on constructors are important to support key spaces: because calling a constructor may create a new object or retrieve an existing one it is important that they do not rely on or change mutable state which may or may not already exist. Programmers are encouraged to write initialisation methods which are called after the constructor returns.

[1]We plan to introduce a class of functions which to not read from or modify mutable state which can be used by constructors, but we have not formalised this yet.

2.5.2 Inheritance Patterns

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 the 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`.

3. EXAMPLE

This section demonstrates Affinity's language features by discussing the implementation of a simple program for creating course schedules (this example is rather well known [4, 5, 10]). The program represents students and courses as objects. Students store their name and id (ids are unique) and the marks they receive for each course. The course stores the name of the course and the number of lectures per week.

We begin by defining a class for students. Student objects store information about a particular person who is enrolled in courses. They have a unique id, a (mutable) name field, and they need to store the grades that the student receives for each course they are enrolled in.

Ignoring grades and unique ids for now we can create a student class:

```
class Student { var name; var id; }
```

This declaration creates a `Student` class with `name` and `id` fields.

Since we know the `id` field should be unique and immutable, we can should make it immutable:

```
class Student { var name; final id }
```

This doesn't prevent multiple students with the same id, but we can use a key space to cause the id field to be keyed, making it unique:

```
Student[university]("alice", "1001");
```

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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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

[1] Java 6 collections api, April 2009.

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

[3] M. Baldoni, G. Boella, and L. van der Torre. Relationships meet their roles in object oriented programming. In *FSEN*, 2007.

[4] S. Balzer, T. R. Gross, and P. Eugster. A relational model of object collaborations and its use in reasoning about relationships. In *ECOOP*, 2007.

[5] G. M. Bierman and A. Wren. First-class relationships in an object-oriented language. In *ECOOP*, 2005.

[6] J. Dolby, M. Vaziri, and F. Tip. Finding bugs efficiently with a sat solver. In *ESEC/SIGSOFT FSE*, 2007.

[7] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA Companion*, 2004.

[8] S. Nelson, D. Pearce, and J. Noble. Changing hashcodes: Objects, initialisation, and collections. Technical Report 2009-03, Victoria University of Wellington, 2009.

[9] K. Østerbye. Design of a class library for association relationships. In *LCSD*, 2007.

[10] D. J. Pearce and J. Noble. Relationship aspects. In *AOSD*, 2006.

[11] J. Soukup. Intrusive data structures. *C++ Report*, 10, 1998.

[12] D. Ungar and R. B. Smith. Self: The power of simplicity. In *OOPSLA*, 1987.

[13] M. Vaziri, F. Tip, S. Fink, and J. Dolby. Declarative object identity using relation types. In *ECOOP*, 2007.