

# Subtyping and Protection in Persistent Programming Languages

Michael Hollins, John Rosenberg and Michael Hitchens

Department of Computer Science,  
University of Sydney,  
Australia.

{mickh, johnr,michaelh}@cs.usyd.edu.au

## Abstract

*Information hiding or encapsulation is a protection mechanism which prevents users from directly accessing certain fields of an object. In many cases, particularly in persistent systems, it may be desirable to provide varying degrees of encapsulation of an object. This paper presents the mechanisms for controlling the encapsulation of objects in the Mozzie programming language. Encapsulation is modelled using the language's structural subtyping mechanism and may be enforced by use of the language's capability mechanism. Capabilities allow for the dynamic control of encapsulation, including the ability to increase or decrease the amount of encapsulation enforced via a certain object reference.*

## 1. Introduction

Persistent systems may consist of large volumes of data which have usually been constructed over time by a community of users. As such the data is valuable and needs to be protected against deliberate and accidental misuse. Protection is required to ensure that users do not misuse shared resources, to protect users from other users and often to protect users from themselves [15].

Persistent systems allow for the possibility of a single model of protection since all data is accessed in a uniform manner. In conventional systems, different protection schemes need to be devised for the different types of data. For example, different mechanisms are required by Unix to protect against illegal access to files compared to those required to protect against illegal access to a process's address space.

Information hiding or encapsulation<sup>1</sup> is a protection mechanism which prevents users from directly accessing certain fields of an object. In many cases, particularly in persistent systems, it may be desirable to be able to

provide varying degrees of encapsulation of an object. For example, a user may create a directory object and wish to retain the ability to add entries to the directory while providing another user with a more restricted view of the object which only allows entries to be retrieved. In addition, it is desirable to be able to dynamically control the level of encapsulation provided.

This paper presents the mechanisms available for controlling the encapsulation of objects in the Mozzie persistent programming language and provides concrete examples of their use. Encapsulation is modelled using the language's structural subtyping mechanism and may be enforced by use of the language's capability mechanism. Capabilities allow for the dynamic control of encapsulation including the ability to increase or decrease the amount of encapsulation enforced via a certain object reference. Mozzie's principle originality comes from the integration of the capability based protection with the object-oriented type system.

Section 2 provides an overview of the Mozzie programming language and describes how encapsulation may be modelled using the subtyping rules of the language. Section 3 briefly describes how persistence is realised in Mozzie. Section 4 provides an example persistent environment written in Mozzie and forms the basis for the examples in the following section. Section 5 presents the mechanisms which may be used to enforce encapsulation and provides examples of their use and Section 6 briefly discusses related work.

## 2. Mozzie

Mozzie is a prototype based object oriented persistent programming language. It is strongly and, for the most part, statically typed. The type system of Mozzie uses structural type equivalence and includes scalar types, such as integers, reals and strings, which display value semantics, and objects which display pointer semantics. The following sections provide an overview of the features of the Mozzie programming language which are relevant to this paper.

---

<sup>1</sup> The term "encapsulation" has been used in the literature to refer to simple data aggregation as in pascal records or as the ability to hide internal detail while providing a public interface to such aggregations. In this paper we will use the term to mean the latter.

## 2.1 Objects

Mozzie's object model is fairly conventional. An object consists of a set of named fields. A field may be either a data field or a procedural field (a method). As with most object oriented languages, objects display pointer semantics.

## 2.2 Object Types

An object type is declared in Mozzie as follows:

```
type Person is object {
  name : string;
  age  : int;
  Name : method(→ string);
  Age  : method(→ int);
};
```

After this declaration the name *Person* is an alias for an object type which contains two data fields, *name* of type *string* and *age* of type *int*, and two methods, *Name* and *Age*. *Name* returns a string and *Age* returns an *int*. Neither method takes any parameters.

A type may also be specified by extending another type. For example:

```
type Student is object Person {
  student_id : string;
  StudentID  : method(→ string);
};
```

The above declaration specifies a new type, *Student*, which contains all the fields of the type *Person* as well as the additional fields, *student\_id* and *StudentID*.

Type equivalence in Mozzie is structural. Two object types are equivalent if they have the same set of field names and for each corresponding field the types are the same. Thus the type *Student* is equivalent to the type *Student2* below:

```
type Student2 is object {
  name      : string;
  age       : int;
  student_id : string;
  Name      : method(→ string);
  Age       : method(→ int);
  StudentID : method(→ string);
};
```

Most statically typed object oriented languages use named typing. eg: C++[17], Java[2] and Eiffel[13]. In such languages, *Student* and *Student2* would not be type equivalent. Modula-3[9] supports structural typing, but includes the shape of the type hierarchy in its definition of the structure of an object type. The effect of this is that *Student2* would still not be equivalent to *Student* since *Student* was formed by extending the *Person* type, whereas *Student2* was not.

Mozzie also supports structural type conformance (structural subtyping) on object types. This allows for variables of different types to refer to the same object (inclusion polymorphism). In order for a variable to refer to an object, the full (dynamic) type of the object must be a subtype of the declared (static) type of the variable. An object type *t2* is a subtype of object type *t1* if *t2* has all the field names of *t1* (possibly more) and all the common fields have the same type.

Thus the types *Student* and *Student2* declared above are both subtypes of the type *Person*. Modula-3 uses a more restrictive rule in which the subtype-supertype relationship must be explicitly denoted. An object type may only have one immediate supertype. For example, in Modula-3, the type *Student* would be a subtype of *Person*, but the type *Student2* would not.

Mozzie's subtyping rule is fairly simple, but it suffices for our purposes of providing language protection mechanisms. The rule is more restrictive than the subtyping rule proposed by Cardelli [6], but avoids the well known difficulties associated with field updates as shown in Connor et al [7].

## 2.3 Object Creation

The code sample below provides an example of the creation of an object in Mozzie:

```
let mick := new Person {
  name := "Michael Hollins";
  age  := 28;
  Name := method(→ string) { name };
  Age  := method(→ int) { age };
};
```

The **let** statement is used to introduce a new variable (in this case the variable *mick*). The type of the variable is inferred from the type of the initialisation expression which must be provided on the right of the '='. The **new** expression is used to create a new object of the specified type. Each of the fields of the object must be initialised, including the method fields. A method is defined by providing a method body between matching curly braces. The return value of a method is the value of the last statement in the body. In this example, each method body consists of only one statement and hence that is the return value of the method.

Normally **new** expressions are wrapped inside a generator procedure as in *makePerson* below:

```

let makePerson :=
proc(n : string; a : int → Person) {
  // return a newly created person
  new Person {
    name := n;
    age := a;
    // .... methods defined as above .....
  }
}

```

The procedure *makePerson* may be used as follows:

```

let paul := makePerson("Paul McCartney", 50);
let john := makePerson("John Lennon", 50);

```

## 2.4 Using Subtyping for Information Hiding

The Mozzie subtyping rules may be used to provide information hiding. For example we may redefine the *Person* type to only include the fields intended to form part of the interface of person objects:

```

type Person is object {
  Name   : method(→ string);
  Age    : method(→ int);
};

```

Next we declare a new type, *ftPerson* (short for *full type of Person*), which extends the type *Person* to include the fields necessary to provide a particular implementation of the *Person* type. For example:

```

type ftPerson is object Person {
  name : string;
  age  : int;
};

```

The *makePerson* routine is then written so that it creates an object of type *ftPerson*, but returns a reference to it of type *Person*:

```

let makePerson :=
proc(n : string; a : int → Person) {
  new ftPerson {
    name := n;
    age := a;
    Name := method(→ string) { name };
    Age := method(→ int) { age };
  }
}

```

Note that *makePerson* is declared to return an object of type *Person*. In fact it creates and returns an object of type *ftPerson*. This is statically type correct since *ftPerson* is a subtype of *Person*.

Now if person object is created using *makePerson*, then the *name* and *age* fields will not be directly accessible:

```

let santa := makePerson("Kris Kringle", 897);
let name := santa.Name(); // OK
santa.name := "Bart"; // ERROR: Fails static check.

```

Using structural subtyping to provide encapsulation as shown above enables more flexibility than is found in most object oriented languages as any level of encapsulation may be achieved. For example:

```

type NamedObject is object {
  Name : method(→ string)
};
let santaAsNamedObject : NamedObject := santa;

type PersonNoMethods is object {
  name : string;
  age  : int;
};
let santaNoMethods : PersonNoMethods := santa;

```

The *santaAsNamedObject* variable may only be used to access the *Name* method of the *santa* object, while the *santaNoMethods* variable may not be used to invoke the methods of *santa*.

In general, any supertype of the full type of an object may be used to provide a restricted view of that object. Whilst the same may be said for any object oriented language, the difference with Mozzie is in the generality of the subtyping rule. Most statically typed object oriented languages, including Modula-3, require any subtype-supertype relationship between types to be explicitly stated. In Mozzie on the other hand, the subtyping relationship between two types is implicit in the structure of the two types.

## 2.5 Dynamic Type Projection

As in many object oriented languages, Mozzie provides an operation for performing explicit dynamic type projections from a supertype to a subtype. For example:

```

let santaAsftPerson := project santa onto ftPerson;
santaAsftPerson.name := "Rudolph";

```

The **project** statement takes an object and a type and performs a dynamic type check to ensure that the full type of the object is a subtype of the specified type. If the dynamic type check fails an exception is raised<sup>2</sup>. **project** is equivalent to Eiffel's conditional assignment operator and Java's explicit cast operator.

The ability to perform projections may be used to circumvent the information hiding described in section 2.4. In a cooperative software development environment this is not seen as a problem as programmers will agree, by convention, not to use projection for this purpose. Such a convention works in the C++ community in which programmers agree not to use explicit casts to access

<sup>2</sup> An alternative typecase form of the **project** statement is provided which prevents exceptions being raised by providing a **default** case which traps errors. This has the same case style format as the Modula-3 **TYPECASE** statement.

encapsulated data. Instead they agree to access objects via their public interface. Note that even if a rogue Mozzie programmer did use projection to break encapsulation, the type system would not be broken as a projection involves a dynamic type check. This is in contrast to C++ in which the (unsafe) type system may be completely circumvented by casting an expression to a completely incompatible type<sup>3</sup>.

In a non-cooperative environment it may be required that the ability to project from a supertype to a subtype be controlled in some way. Mozzie provides mechanisms for enforcing the encapsulation of objects when required. These are described in section 5.

### 3. Persistence

The ability to coerce values from subtypes to supertypes and back again allows for the creation of a persistent store of objects built around the subtyping rules. Such a store uses persistence by reachability in which all objects which are reachable from the root (or roots) of persistence will persist. The root of persistence is a value stored in a known location from which all other objects may be found by the traversal of object references.

To provide maximum generality, the root of persistence must be able to store a value of any object type. This is achieved in Mozzie by giving the root the following type:

```
type Any is object { }
```

*Any* is the type of an object with no fields. All object types conform to this type. Thus, if we have a variable *root* declared as:

```
root : Any;
```

and a variable *john* of type *Person*, then to set the value of the persistent root to *john* we perform an assignment<sup>4</sup>:

```
root := john;
```

To retrieve the object from the persistent root we use an explicit dynamic type projection to convert from the type *Any* to a more specific type. For example:

```
let rootPerson := project root onto Person;
let name := rootPerson.Name();
```

### 4. An Example

In this section persistent environment modelled on Napier's *env* facility [14] is developed. This example is provided to give a basis for the examples in the sections

<sup>3</sup> Type safe runtime type projection and interrogation have recently been added to the C++ language definition. The type system remains un-safe, but the user now has the option to perform casts in a type safe manner if required.

<sup>4</sup> Of course in a real system the root of persistence would refer to something more interesting than a single person object such as an object of type *Env* (see section 4).

which follow. Note that for the sake of simplicity the examples do not consider the handling of errors.

An *Environment* is analogous to a directory in a traditional file-based system. It is used to map string names to objects. Environments (like directories) may be nested by placing an environment within another environment. The type of an *Environment* is specified below.

```
type Any is object { };
type Env is object {
  Add : method(string, Any);
  Get : method(string → Any);
  Drop : method(string);
  Open : method(→ EnvIterator)
};
```

The *Add* method takes a name and an object and places the object into the environment. The *Get* method is used to retrieve objects by name. *Drop* removes the specified object from the environment. The *Open* method is used to iterate over the names in an environment. *Open* returns an object of type *EnvIterator* specified below:

```
type EnvIterator is object {
  Reset : method(); // reset to start of list
  Next : method(); // go to next element
  CurrentName : method(→ string);
  CurrentObject : method(→ Any);
  isEnd : method(→ bool);
};
```

The *CurrentName* method returns the name of the current entry. *CurrentObject* returns the current object. *isEnd* returns *true* if there are no more elements.

An implementation of environments is provided by extending the *Env* type to include fields which will form the encapsulated data:

```
type ftEnv is object Env {
  // an array implementation is used for simplicity
  numEntries : int;
  names : vector[string]; // array of entry names
  entries : vector[Any]; // array of entries
};
```

and providing an associated *make* procedure<sup>5</sup>:

<sup>5</sup> Structural subtyping makes it easy to supply different implementations of a particular type. Several implementations of the *Environment* type may be provided, for example, by specifying different encapsulated data (*ftEnvironment* types) and corresponding *make* procedures.

```

let makeEnv := proc(→ Env)
{
  // create new environment
  let newEnv := new ftEnv {
    numEntries := 0; // no elements yet
    names := vector 1 to 100 of "";
    entries := vector 1 to 100 of nil;
    Add := method(nm : string; obj : Any) {
      numEntries := numEntries + 1;
      names@numEntries := nm;
      // see footnote 6
      entries@numEntries := obj
    }
    ... other method definitions ...
  };
  newEnv // return newly created environment
};

```

Here a local variable, *newEnv*, is used to refer to the environment object to be returned by *makeEnv*. Although not needed for this example, the variable will prove useful in examples later in the paper.

The following code provides example usage of *makeEnv*.

```

let rootEnv := makeEnv();
root := rootEnv;
rootEnv.Add("mickh", mick);
rootEnv.Add("Santa", santa);

```

An environment *rootEnv* is created and assigned to the root of the persistent store. Next, two person objects are added to the *rootEnv* environment.

The following code shows how an environment entry may be retrieved and accessed at a later time:

```

let rootEnv := project root onto Env;
let mickh := rootEnv.Get("mickh");
let mick := project mickh onto Person;
IO.WriteString("name is " ++ mick.Name());

```

Finally, the elements in an environment may be iterated over by calling the *Open* method:

```

let it := rootEnv.Open();
IO.WriteString("List of Items in root env:\n");
while(~it.isEnd()) {
  IO.WriteString(it.CurrentName());
  it.Next()
}

```

## 5. Controlling Dynamic Type Projection

As stated previously, the ability to perform projections may be used to circumvent encapsulation. For example, in

the previous section, instead of projecting *rootEnv* onto the *Env* type via:

```

let rootEnv := project root onto Env;

```

the programmer (armed with the knowledge of the full type of *rootEnv*) could have projected onto the full type:

```

type ftEnvironment is ...
let rootEnv := project root onto ftEnv;

```

and proceeded to access fields which were not intended to be directly accessed (at least by that user).

Mozzie provides facilities which allow for the encapsulation by subtyping technique described in section 2.4 to be enforced when required. Central to these is the *capability* mechanism described in the next section.

### 5.1 Capabilities

All object references in Mozzie are capabilities. A capability contains a reference to the object being protected and a set of access permissions. Capabilities are protected from arbitrary modification or forgery. One object may have many capabilities referring to it, each of which may specify a different set of access permissions for that object. Table 1 shows the information stored by each capability.

**Table 1: The structure of a capability.**

Item	Description
object	reference to object
field rights	true/false for each field of the object

The *object* field is an implementation dependent reference to the object controlled by the capability. The *field rights* specify which of the object's fields may be accessed via this capability. A particular field may only be accessed if the caller provides a capability which has the corresponding *field rights* entry set to true.

When an object is created, a reference to a capability is returned. This capability contains the maximum possible rights for objects of that type. Assignment of objects (and parameter passing) involves the copying of the reference to the capability. The capability itself is not copied. Hence, more than one variable may reference an object via a particular capability.

An attempt to access a field or execute a capability operation via a capability which does not contain the relevant permission results in the raising of an exception.

Due to space limitations, a full description of the Mozzie capability mechanisms is not provided here. Refer to [10] for a more detailed description.

### 5.2 Capability Operations

This section describes the two capability operations which are strictly relevant to this paper. The **derive**

<sup>6</sup> The @ symbol is used in mozzie to index arrays (vectors).

operation is used to create a copy of an existing capability. It makes a complete copy of the original capability and returns a reference to the new capability. The new capability will contain the same rights as the original capability. For example:

```
let peopleEnv2 := derive peopleEnv;
```

The type of the **derive** expression is the same as the type of the supplied object. In this case the type will be *Env*.

The **restrict** operation is used to restrict the field rights of a particular capability to just those fields which are part of a particular type. It takes two parameters. The first is a reference to the capability whose permissions are to be reduced. The second parameter is the type to which the field rights in the capability are to be restricted. For example:

```
restrict peopleEnv2 to Env;
```

### 5.3 Enforcing Encapsulation

The following code shows how the *makeEnv* procedure may be rewritten to use **derive** and **restrict** to enforce the encapsulation of the environment objects it creates. The highlighted section shows the code which has been inserted:

```
let makeEnv := proc(→ Env)
{
  let newEnv := new ftEnv { ... };
  let ret := derive newEnv;
  restrict ret to Env;
  ret
};
```

Note that the user is still permitted to perform the dynamic projection as in:

```
let myEnv := makeEnv();
let myftEnv := project myEnv onto ftEnv;
```

The projection will succeed. It is only when the user attempts to access the (previously) encapsulated fields of the object that an insufficient access permission error will be detected and an exception raised. eg:

```
myftEnv.numEntries := 50;
```

The reason for delaying the check is that the type of a reference and the permissions available via that reference are orthogonal properties of the reference. The permissions available via a particular reference (capability) may be altered (increased or decreased) at any time during the lifetime of that reference. So, it is possible for the permissions to be unavailable at the time of the **project**, but available at the time of actual object access.

However, Mozzie does provide an operation, **allows**, which checks whether access to all the fields in a type are permitted via a certain capability. For example:

```
if myftEnv allows ftEnv then . . . .
```

### 5.4 Accessing Encapsulated Data

The capability mechanisms make it possible to enforce encapsulation via one reference whilst allowing encapsulated data to be accessed via another reference. For example, the following code provides a new implementation of *makeEnv* in which *newEnv* is stored away somewhere for later use:

```
let makeEnv := proc(→ Env)
{
  let newEnv := new ftEnv { ... };
  Store(newEnv); // store newEnv somewhere
  . . . .
  ret
};
```

The set of *newEnv* capabilities may be used, for example, to provide binary methods. That is, methods which need to act on the encapsulated data of more than one object.

The ability to be able to enforce encapsulation via some references whilst enabling access to encapsulated data via other references as described above can be extremely important in a persistent system. The need for such a mechanism arises from the fact that data is only ever stored in a single format. In a conventional (non persistent) system such a mechanism is not as important since encapsulation is effectively broken in an uncontrolled fashion when the information contained in objects is saved to some external, long lived format such as a flat file or a relational database.

### 5.5 Tags

The previous section provided an example of how capabilities of varying strengths may be created for a particular object. Each time *makeEnv* is called two capabilities are created. The original capability is stored away somewhere for later use while the second is returned to the caller and is their responsibility to manage. Management (and efficient retrieval) of the original capabilities may become difficult as the number of calls to *makeEnv* may be quite large.

Mozzie provides a tag mechanism which alleviates this management problem. A *Tag* allows a capability with superior rights to be retrieved via a capability with lesser rights. The basic idea behind tags is that a capability *C* may be provided (tagged) with a tag *T* so that *C* may be later retrieved from any capability derived from *C* (or any capability derived from a capability derived from *C* and so on) by providing *T*. The power of tags comes from the fact that a single tag may be used to tag many capabilities for many different objects.

Tags are created by calling the *makeTag* polymorphic procedure:

```
makeTag : proc[T](→ Tag[T])
```

where *Tag* is the following parameterised type:

```

type Tag[T] is object {
  tag : method(T);
  retrieve : method(Any → T);
};

```

For example:

```

let envTag := makeTag[ftEnv]();

```

declares a variable *envTag* which refers to a tag which may be used to tag capabilities of type *ftEnv*.

The *tag* method sets up a relationship between a tag and a capability so that the tag may be used later to retrieve the capability from one of its descendants. For example:

```

envTag.tag(newEnv);

```

The *retrieve* method may be used later to retrieve the original capability given any of its descendants. *retrieve* returns a value with the same type as was provided to *makeTag*:

```

let env := envTag.retrieve(someEnv);

```

*env* above is of type *ftEnv* since the type of *envTag* is *Tag[ftEnv]*. The object provided to *retrieve* may have any type.

As with the **project** statement, dynamic failure of the *retrieve* operation results in the raising of an exception. *retrieve* fails if the tag was not used to tag the object supplied to *retrieve*.

## 5.6 Tags Example - Binary Methods

A *binary method* is one which takes two (or more) parameters of the same type. The provision of binary methods in object oriented languages presents two difficulties. Firstly, "typing binary methods in the presence of inheritance" and secondly, gaining "privileged access to object representations" [5]. This section provides an example of using tags to solve the latter problem.

In this example, an object, *maker*, will be created which may be used to create new environment instances. In this respect, *maker* will be playing the role that a class plays in a class based language. The type of the *maker* object is specified below.

```

type EnvMaker is object {
  makeEnv : method(→ Env);
  SameSize : method(e1, e2 : Env → bool);
};

```

*makeEnv* is used as before to create new environment objects. The *SameSize* method is an example of a binary

method. It takes two environment objects and returns **true** if both environments have the same number of entries<sup>7</sup>.

The full type of the environment maker object, *ftEnvMaker*, also includes a tag, *envTag*, which will be used to tag the environment objects created by *makeEnv*:

```

type ftEnvMaker is object EnvMaker {
  envTag : Tag[ftEnv]
};

```

The *maker* object itself is created as follows:

```

let maker := new ftEnvMaker {
  envTag := makeTag[ftEnv]();
  makeEnv := method(→ Env)
  {
    let newEnv := new ftEnv
      ....
  };
  envTag.tag(newEnv);
  let ret := derive newEnv;
  restrict ret to Env;
  ret
}
SameSize := method(e1, e2 : Env → bool) {
  let E1 := envTag.retrieve(e1);
  // extract more powerful capability
  let E2 := envTag.retrieve(e2);
  // extract more powerful capability
  E1.numEntries = E2.numEntries
}
};

```

*makeEnv* uses the tag *envTag* to tag each of the capabilities it creates. The *SameSize* method uses **retrieve** to gain access to more powerful capabilities for each of the environment objects passed in. These more powerful capabilities may then be used to access the encapsulated data and compute the method result.

After the creation of the *maker* object, access to the *envTag* field needs to be protected. This is achieved by providing users with an encapsulated view of the object:

```

let sharedMaker : EnvMaker := derive maker;
  // derive followed by an explicit type coercion
restrict sharedMaker to EnvMaker;

```

The first line above creates a new variable, *sharedMaker*, of type *EnvMaker*, which refers to a derived capability for the *maker* object. The second line restricts the access rights in the capability referred to by *sharedMaker* to just

<sup>7</sup> Of course such a routine could be written without having to access the encapsulated data of the environment objects by creating an *EnvIterator* object for each environment and iterating through counting the number of entries in each. However, our *SameSize* routine will almost certainly be more efficient as it directly accesses the *numEntries* field in each environment. Also, there are many examples of binary methods for which such access is essential due to the encapsulated data containing fields whose values cannot be calculated from the object interfaces.

those fields which belong to the type *EnvMaker*. That is, access is restricted to just using the *makeEnv* and *SameSize* methods.

The above example shows how tags may be used to achieve the equivalent of Java and C++ class level (static) methods. Such methods are allowed to access the encapsulated data of all instances of that class.

Mozzie provides a more flexible mechanism, however. For example, it is possible for two different users (*A* & *B*) to use different *maker* objects (*makerA* & *makerB*) to create their respective environment objects. The objects created by *makerA* will be type compatible with the objects created by *makerB* (they are all of type *Env*); however, they will have been tagged by different tags, thereby denying user *A* access to the encapsulated data of the environment objects created by user *B* and vice versa.

In most OO languages the type of an object and the class of an object are one and the same thing. Hence such flexibility is not achievable. If *A*'s environment objects are created by a different class to *B*'s environment objects, the objects thus created will not be type compatible.

The principle drawback to using tags instead of some existing static mechanism is that, being a dynamic mechanism, a run-time check is required. In general, it is anticipated that the check required will not entail a significant run-time overhead.

### 5.7 Creating Binary Methods for Existing Objects

This section shows how tags may be used to create additional binary methods for objects already in existence. Such functionality is not provided by traditional object oriented protection mechanisms.

Suppose that the provider of the *maker* object would like to provide an additional binary method, *Smaller*, which takes two environment objects and returns **true** if the first environment has less entries than the second.

```
Smaller : method(e1, e2 : Env → bool)
```

To implement *Smaller* access is needed to the tag which was used by *makeEnv* to tag each of the objects which it created. This tag is available to the provider of *sharedMaker* as part of the *maker* object. ie:

```
maker.envTag;
```

A new object may be created whose interface consists of the *Smaller* method and whose encapsulated data consists of the tag *maker.envTag*. As usual the type of the object which will form the interface of the new object is specified:

```
type CompareEnvs is object {
  Smaller : method(Env, Env → bool)
};
```

Next the full type is specified by extending the *CompareEnvs* type:

```
type ftCompareEnvs is object CompareEnvs
{
  envTag : Tag[ftEnv]
};
```

Lastly, the actual object is created:

```
let compareEnvs := new ftCompareEnvs {
  envTag := maker.envTag;
  Compare := method(e1, e2 : Env → bool)
  {
    let E1 := envTag.retrieve(e1);
    let E2 := envTag.retrieve(e2);
    E1.numEntries < E2.numEntries
  }
};

// create a restricted view of the compareEnvs
// object for public use
let sharedCompareEnvs : CompareEnvs :=
  derive compareEnvs;
restrict sharedCompareEnvs to CompareEnvs;
```

### 5.8 Tags as objects.

The Tag mechanism is actually an abstraction built on top of the Mozzie object model. Tags are in fact just objects. As such, they are referred to by capabilities and the use of the tag operations (*tag* and *retrieve*) may be controlled in the same manner as access to any object fields are controlled. For example, a user *U1* may derive a new capability *T2* for a tag *T1* and pass that capability to another user, *U2*. At a later time, *U1* may revoke *U2*'s use of the tag by reducing the access rights in *T2*.

### 5.9 Using Tags for Dynamic Type Projection.

As an aside, notice that the *retrieve* operation takes a value of type *Any* and returns a value of some subtype. The operation is able to return a value of the appropriate type without having to perform the dynamic type check required by the **project** statement. This is because it is known that if the tag matches one of the tags used to tag the object, then the type of the tag must be a valid supertype of the full type of the object and a further dynamic type check is unnecessary. It is expected that in general the *retrieve* operation will be much faster than a full structural type check. Thus a program may use the tag mechanism to perform efficient run-time type checking by tagging objects which it knows will later require dynamic type projection.

### 5.10 Implementation of the Open() method

This section describes the implementation of the *Open* method of the *Env* type as it provides another example of the use of the capability mechanisms to enforce

encapsulation. First the full type of the iterator is specified by extending the *EnvIterator* type.

```

type ftEnvIterator is object EnvIterator {
  env : ftEnv; // the env we are iterating over
  current : int // where we are up to
};

```

The implementation of the *Open* method is shown below:

```

Open := method(→ EnvIterator) {
  let iterator := new ftEnvIterator {
    env := self; // iterate over the current object
    current := 1;
    Reset := method() { current := 1 };
    Next := method() { current := current + 1 };
    CurrentName := method(→ string) {
      env.names@current
    };
    CurrentObject := method(→ Any) {
      env.entries@current
    };
    isEnd := method(→ bool) {
      current > env.numEntries
    }
  };

  // return a restricted reference to the iterator
  restrict iterator to EnvIterator;
  iterator
}

```

The point to note is that the iterator object needs unencapsulated access to the environment object in order to perform its function. It stores the unencapsulated view in the *env* field. It is therefore important for *Open* to return capabilities for *EnvIterator* objects which cannot be used to access the encapsulated data of the iterator objects. Otherwise the encapsulation of an environment could be broken by creating an iterator for the environment and accessing the encapsulated data indirectly via the iterator.

In C++, a similar effect can be achieved by making the *EnvIterator* class a friend of the *Env* class. This gives all of the methods of the *EnvIterator* class the right to access the encapsulated data of all instances of the *Env* class. Notice that this is a more general condition than is actually required in this case. What is actually required is for the methods of a particular object to be able to access the encapsulated data of another object. In *Mozzie* it is possible to express such a protection requirement between two individual objects as shown in the example above.

## 6. Related Work

The idea of including access control mechanisms within a programming language is not new. Jones and Liskov for example, presented an extension to the type

system of object-based programming languages to support the expression of access constraints on shared data [11]. In retrospect, the extensions can be viewed as adding structural sub-typing to the type-system. The Jones and Liskov protection mechanisms only provide support for static control of access permissions and are really intended as a software engineering tool rather than as a mechanism for providing support for dynamic access control of persistent objects.

Capabilities have existed in numerous forms, principally in operating systems, since the 1960s. Examples are Amoeba [18], Monash [1], Monads [16] and Grasshopper [8]. As stated earlier, *Mozzie's* principle originality comes from the integration of the capability based protection with the object-oriented type system. This allows for the succinct expression of dynamic protection requirements within the language itself. Linden [12] provides a good overview of the use of capabilities to support system security and reliability

Existential types, such as Napier's abstract data types [14] and Modula-3 opaque types, provide a mechanism for implementing binary routines. Values of the abstract witness type may be freely passed around, but access to the underlying concrete type of the values is restricted to the set of routines defined as part of the definition of the existential type. These routines may take an arbitrary number of values (of the abstract type) as parameters and subsequently access the concrete representations. As with the Jones and Liskov mechanisms, however, existential types provide a static mechanism and do not permit the creation of additional binary routines on pre-existing values as is possible with the *Mozzie tag* mechanism.

The implementation of existential types is usually achieved by the generation of some unique "magic number" which may be used to check for type equality between two abstract values. Modula-3, for example, uses branded types to protect its opaque types. A *Mozzie tag* may be viewed as such a brand except that it is exposed to users. Users may use the tag to create new binary operations which act on pre-existing values and, for that matter, binaries which act on objects of different types. The trade off is that exposing the magic number may lead to misuse of its powers if left in the wrong hands.

*Mozzie's tag* mechanism has much in common with the *extended unions* or *wotsits* of the Ten15 system [3]. The primary difference is that tags are tied to the capability system which means that they can be used to tag any capability for an object. This means that tags may be used to provide access to a subset of the object's fields. Ten15 provides an all or nothing mechanism (A *wotsit* always provides access to the entire object.) and as such does not provide the same flexibility.

## 7. Conclusion

In this paper we have shown how structural subtyping may be used to model encapsulation and have presented language mechanisms which enable such encapsulation to be controlled dynamically. The language's object-oriented type system is used in synergy with the capability system to provide a model of protection which is both powerful and easy to manage. Example usage of the mechanisms has been provided.

The language provides a means by which an object may be encapsulated via one reference whilst remaining unencapsulated via another. The need to be able to access encapsulated data in a persistent system has been described. In a conventional (non-persistent) system, such a mechanism is not as important since encapsulation is effectively broken in an uncontrolled fashion when the information contained in objects is saved to some external, long lived format.

An initial version of a compiler for Mozzie has been completed. The compiler resides on top of the St Andrews persistent object store [4]. Future work will concentrate on experimentation with the mechanisms which Mozzie provides and comparison with existing languages. It is hoped that this work will feed back into the language design. In particular, it is hoped that the mechanisms will be reduced to a core set of mechanisms sufficient for providing the types of protection described in this paper.

Future work will also consider the performance impacts of the additional run-time checks introduced by capabilities. It is hoped that compiler optimisation techniques can be used to improve overall performance. In particular, caching techniques may be used to amortize the cost of capability checks.

## 8. Acknowledgments

This project is supported by Australian Research Council Grant A49330060. The authors would like to thank Ron Morrison, Graham Kirby and Richard Connor from the University of St Andrews for helpful comments during this project.

## 9. References

- [1] Anderson M, Pose R D & Wallace C S. "A Password-Capability System". *The Computer Journal*, Vol 29, No 1, pp 1-8. 1986.
- [2] Arnold K & Gosling J. "The Java Programming Language". Addison Wesley, 1996.
- [3] Blanchard T. "An Efficient Database Language". PhD Thesis, University of York, Department of Computer Science, 1993.
- [4] Brown A L & Morrison R. "A Generic Persistent Object Store". *Software Engineering Journal* 7, 2 pp 161-168, 1992.

- [5] Bruce K, Cardelli L, Castagna G, Leavens G T, Pierce B & The Hopkins ObjectsGroup. "On Binary Methods". *Theory and Practice of Object Systems*, John Wiley & Sons. Volume 1, Number 3, 1995.
- [6] Cardelli L. "A Semantics of Multiple Inheritance". *Lecture Notes in Computer Science*, vol 173, Springer Verlag, pp 51 - 67, 1984.
- [7] Connor R C H, McNally D & Morrison R. "Subtyping and Assignment in Database Programming Languages". In *Proc, 3rd International Workshop on Database Programming Languages*, Morgan Kaufman, pp 363- 382, 1991.
- [8] Dearle A, di Bona R, Farrow J, Henskens F, Hulse D, Lindstrom A, Norris S, Rosenberg J, & Vaughan F. "Protection in Grasshopper: A Persistent Operating System". *Proceedings, 6<sup>th</sup> International Workshop on Persistent Object Systems*, France. Springer Verlag, 1994.
- [9] Harbison S P. "Modula-3". Prentice Hall, 1992.
- [10] Hollins M, Rosenberg J & Hitchens M. "Language Mechanisms for Protecting Persistent Data". *Proceedings of the 19th ACSC Conference*, Melbourne, Australia, January 31 - February 2 1996.
- [11] Jones A K & Liskov B H. "A Language Extension for Expressing Constraints on Data Access". *Communications of the ACM*. May 1978. Volume 21, Number 5, pp 358-367.
- [12] Linden T A. "Operating System Structures to Support Security and Reliable Software". *ACM Computing Surveys*, Vol 8, No 4, December 1976.
- [13] Meyer B, "Eiffel: The Language". Prentice Hall, 1992.
- [14] Morrison R, Brown A L, Connor R C H, Cutts Q I, Dearle A, Kirby G, & Munro D. "The Napier88 Reference Manual (Release 2.0)", University of St Andrews, Research Report CS/93/15. 1993.
- [15] Morrison R, Brown A L, Connor R C H, Cutts Q I, Kirby G, Dearle A, Rosenberg J & Stemple D. "Protection in Persistent Object Systems", *Proc, International Workshop on Computer Architectures to Support Security & Persistence of Information*, Bremen, 1990. Springer Verlag. pp 48-66.
- [16] Rosenberg J & Abramson D A. "MONADS-PC: A Capability Based Workstation to Support Software Engineering". *Proceedings of the 18th Hawaii International Conference On System Sciences*, 1985, pp 222-230.
- [17] Stroustrup B. "The C++ Programming Language". Addison-Wesley Publishing Company. 1991.
- [18] Tanenbaum A S, van Renesse R, van Stavern H, Sharp G J, Mullender S J, Jansen J & van Rossum G. "Experiences with the Amoeba Distributed Operating System". *Communications of the ACM*, Vol. 33, pp 46-63. Dec 1990.