1V-01 A Language for Mobile Objects with Dynamic Grouping

Yasushi Kambayashi
University of Toledo

Masaki Takahashi
Keio University

Munehiro Takimoto
Science University of Tokyo

Ken'ichi Harada
Keio University

1 Introduction

The design of a language that supports mobile objects with dynamic grouping is presented. It is becoming more common to see mobile objects in distributed computing environments. The mobile object is a convenient means to utilize resources residing in remote sites. Unlike traditional remote procedure calls that require transmission every time they are executed, self-containing mobile objects can perform great deal of work once they are transmitted to remote sites. Since mobile objects can interact with site environments, they can even customize their behaviors according to those environments.

Mobile objects are especially useful in environments where networks are frequently disconnected, (e.g. notebook computers), or to avoid high transmission cost, (e.g. via international telephone-lines).

Computational models that support such language constructs exist, such as Mobile Ambients[2] of Luca Cardelli and Distributed Join-Calculus[3] of INRIA. Our objective is to design a language based on the same philosophy as Distributed Join-Calculus and support dynamic grouping. Such a language should also prove useful for software component reuse.

2 Design Principles

Our language is based on the well-known object oriented functional language O'Caml[6], which is derived from the efficient functional language Caml[5]. O'Caml has a class-based object creation mechanism with strong typing facilities. Since O'Caml provides the basic construct for creating objects as well as functions, our design concentrates on mobility of the objects and dynamic grouping, so that we can preserve the semantics of O'Caml as much as possible. The first goal, mobility, is satisfied by our previous language, DOCaml[4].

3 Mobile Objects

Our language employs the distributed lexical scope proposed by Cardelli and implemented in Obliq[1]. All bindings of free variables are solved at compile time. For example, after an object, obj1, moved from site 1 to site 2, it calls a method in obj0 as shown in figure 1. Since there are two objects that share the name obj0 — one in site 1, and the other in site 2 — we have to solve which object is targeted. In the distributed lexical scope policy, the method in obj0 in site 1 is executed through proxy. In this respect, the semantics of the objects are preserved over the distributed environment.

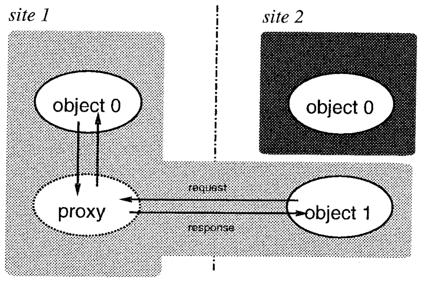

Figure 1.

The migration of objects is accomplished through cloning and re-binding by the remote application of the cloning function:

```
class point x_init =
  val mutable x = x_init
  method get    = x
  method add a  = x <- x + d
end;;

let o = new point 10;;
let o = clone o at site2;;
```

This point class has one mutable instance variable x, and two methods, get and add, as members. The initial value for the instance variable is given through the class parameter x_init.

In the first let-expression, applying the function new with argument 10 to the class point creates an object o. In the second let-expression, the created object migrates to site 2. Migration is achieved not through serialization, but cloning. The function clone is a special form and takes the object type as the argument, and makes duplication and returns it.

This function application, clone o, is executed actually in site 2 so that the duplicated object resides in site 2. The return value is transmitted to the original location, site 1, and is bound to o. Now o is the proxy of the object in site 2.

By putting the moving method in the class description, an object can move itself. Such a method must have the remote application of the function described above. Since each object has site information, this method must update that information, as follows:

```
Class x() as self =
  ...
  method go place = self @= clone self at place
  ...
end;;
```

4 Dynamic grouping

One of the innovative features of this language is the dynamic grouping. The dynamic grouping allows objects created in different sites to communicate with each other, and can combine themselves into yet another larger object. Through this mechanism, we can define objects as components in the software engineering sense, and then send them in various combinations to remote sites, and make them combine into specific objects to accommodate the specific site requirement. For example, we can provide different interface objects in different sites. Then we can send the same core functional object to each site and make them combine into one object that interacts to the site environment, providing the identical service. Or we can send a new feature as an object to the larger object already residing in the remote site so that we can update the service object dynamically.

In order to combine, objects are constructed as a hierarchical structure. In other words, objects are combined by placing one object into another object. For this purpose, this language allows us to specify an object in which the function application is completed. Therefore, a site is considered as a special object in this respect. In the following let-expression, `obj0` moves into `obj1`; then the combined object, `obj1`, further moves into the other object, `obj2`.

```
let obj0 = new classA ();;
let obj1 = new classB ();;

let obj0 = clone obj0 at obj1;;
let obj1 = clone obj1 at obj2;;
```

The special form `clone` duplicates not only the object given as argument but also objects those which that object includes as shown in the figure 2.

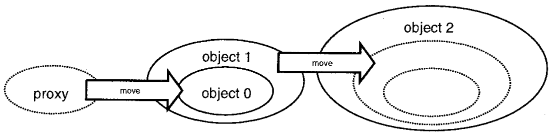

Figure 2.

5 Interaction

Interactions to objects from the outer-world are performed by the method invocation protocol defined in O'Caml as follows:

```
o#get;;
o#add 2;;
o#get;;
```

The expression `o#get` invokes the method `get` in the object `o`. The syntax doesn't distinguish whether the object exists in the local site or remote site. If the object resides in the remote site, the run-time system converts it into a remote function application. The usual function application is executed otherwise. The first expression and the third expression return 10 and 12 respectively. The second expression doesn't return anything (return "unspecified").

Similar to the inheritance convention, the mobile objects can invoke methods in the object that includes those mobile objects. This is achieved through self-reference. Dynamic grouping is not inheritance, but it causes a hierarchy of objects. Therefore, once an object moves into another object, the object acquires the scope of the including object.

Since we cannot guarantee that moving objects always find the appropriate methods in the destination objects, the moving objects that have method invocations, those methods are expected to be defined in the destination, must carry exception handling functions.

6 Conclusion

A design of a functional object oriented language with dynamic grouping is presented. Through the dynamic grouping, we can accomplish software component reuse in distributed and dynamic environments. Each mobile object should be a small software component with a single function. They are supposed to be built in different locations and in different time periods. In order to accomplish a certain task, we can send a set of mobile objects from various sites to the site in which the work should be accomplished. Those sent objects combine themselves into a larger object and execute the task. Since each site has a different environment, and requires different user-interface, such dynamic software composition should be useful in real world software engineering practice.

References

[1] Luca Cardelli, **A Language with Distributed Scope**. *22nd ACM Symposium on Principles of Programming Languages* , pp.286-297, 1995.

[2] Luca Cardelli and Andrew D. Gordon, **Mobile Ambients**. *Proc. European Joint Conference on Theory and Practice of Software*, pp.140-155. 1998.

[3] Cedric Fournet, Georges Gonthier, Jean-Jacques Levy, Luc Maranget and Didier Remy, **A calculus of mobile agents**. *CONCUR'96:Concurrency Theory*, pp.406-421, 1996.

[4] Yasushi Kambayashi, Munehiro Takimoto, Yasushi Kodama, Kenichi Harada, **A Higher-Order Distributed Objective Language**. *Proc. International Symposium on Future Software Technologies*, pp.241-246. 1997.

[5] Xavier Leroy, **Manuel de Reference du Langage Caml.** InterEditions, 1993.

[6] Xavier Leroy, **The Objective Caml System Release 1.07**. *Documentation and User's Manual, Institut National de Recherche en Informatique et Automatique*, 1997.