Agora: Reintroducing Safety in Prototype-based Languages

Wolfgang De Meuter, Tom Mens, Patrick Steyaert
{ wdmeuter@vnet3 | tommens@is1 | prsteyae@vnet3 }.vub.ac.be
Programming Techonology Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium

ABSTRACT. Prototype-based languages are often described as being more flexible and expressive than class-based languages. This greater flexibility makes prototype-based languages well-suited for rapid prototyping and exploratory programming, but comes with a serious loss of safety. Examples of this are the encapsulation problem and the prototype corruption problem most prototype-based languages suffer from. These problems preclude prototype-based languages from being widely used. We propose a prototype-based language that eliminates these problems and thus reintroduces safety in prototype-based languages.

1. Introduction

Usually, prototype-based languages allow a wide variety of operations to be performed on objects. Examples of such operations are message sending, parent assignment, adding a slot to an object and cloning an object [Ungar&Smith87] [Dony&al.92] [Taivalsaari93]. Such operations are often designed without a general theory behind them. For this reason Agora [Steyaert&al.93] [Codenie&al.94] was initially conceived as a prototype-based language kernel based on objects and message sending alone (in the same way as pure functional languages are based on functions and function application alone). The original idea was to add other operations later on in an incremental fashion, in order to precisely determine which object-oriented features behave orthogonal and what kind of semantic constructions (in the implementation) are needed to support them. However, as the Agora project evolved, it became clear that besides message sending no additional operations are needed to obtain a full-fledged object-oriented language.

We will argue that this simple message sending paradigm results in a lucid prototype-based programming language that is much safer than other prototype-based languages. It features a combination of controlled inheritance, controlled cloning and controlled reflective capabilities without neglecting the achievements of other prototype-based languages (such as dynamic object extension). It is the way these features are controlled by the language that makes Agora safe. We claim that this safety is an important prerequisite for prototype-based languages to become a viable alternative for the widely used class-based languages.

2. Agora Language Definition

2.1 Agora = Objects + Messages

In Agora, user defined objects can be created by simply putting a sequence of valid message expressions between a pair of brackets. The semantics of such an "ex nihilo created object" is to create a new object and to evaluate each of its message expressions in the context of that new object.

Agora features two kinds of messages expressions: *ordinary messages* and *reifier messages*. The evaluation strategy for *ordinary messages* is to send the message to the evaluated receiver with the evaluated arguments. This is much like function application in an applicative order functional language like Scheme. *Reifier messages* on the other hand can be seen as messages whose receiver and arguments are not necessarily evaluated. They thus correspond to the notion of reifier functions in 3-Lisp [Smith82] or special forms in Scheme. In the same way as special forms are 'special functions' that are intercepted by the Scheme interpreter, reifier messages can be seen as 'special messages' that are intercepted

by the Agora interpreter. The Agora interpreter recognizes reifier messages by their boldfaced appearance.

An example of a reifier message is the **`variable:`** message, which should be sent to an identifier and whose argument can be any valid Agora expression. The currently chosen implementation of the **`variable:`** message will evaluate its argument and will declare its receiving identifier in the 'current' object. This idea is illustrated in the following code fragment that defines an ex-nihilo created object containing a single instance variable `x` with initial value `10`.

```
[ x variable: 10 ]
```

If we would employ the evaluation strategy for ordinary messages here, a "`variable x not declared`" error would occur.

Like special forms make up the concrete syntax of Scheme, reifier messages define the particular flavour of Agora. The following section describes a subset of the currently implemented reifiers. An elaborate description can be found in the Agora user manual [DeMeuter96].

2.2 Some reifier messages

Methods can be declared in Agora by sending the **`method:`** reifier to a pattern. The argument of the **`method:`** reifier should be a valid Agora expression constituting the body of the method. Methods can be defined **`functional`** or **`imperative`** depending on whether or not they return a result. This corresponds to the difference between functions and procedures in Algol like languages.

Like Self [Ungar&Smith87], Agora is a slot-based language, meaning that its variables are accessed through read and write accessor slots. Hence, a variable `x` defines a couple of method slots `x` and `x:` to read, respectively write the variable. As indicated above, variables are declared using the **`variable:`** reifier.

Inheritance is achieved by yet another kind of methods called **`mixin`** methods [Steyaert&al.93]. Upon invocation, mixin-methods extend the receiving object with the contents of their method body. When a mixin-method is declared **`functional`**, it returns a new object whose parent object is the receiver of the mixin-method. Functional mixin-methods can thus be used to create different views on objects. The usual inheritance rules govern the relation between the parent and each view. When a mixin-method is declared **`imperative`**, the receiving object and all its descendants will be destructively changed with the contents of the mixin-method. This can for example be used to turn an entire hierarchy of black and white graphical objects into a coloured hierarchy in one stroke.

Cloning is accomplished through **`cloning`** methods. These are methods whose body is executed in a shallow clone of the receiver. By definition, cloning methods are always functional, and their result is the clone of the receiver in which they were executed. The body of the cloning method can contain initialisation code.

Every method slot can be declared either **`public`** or **`local`**. Public slots are visible to everyone while local slots are only visible to the object itself.

All these features have been used in the bank account example below. While everyone can deposit money on an account, a client can only withdraw money from the account if she provides the correct user password. On request of a client, a clerk can create new bank accounts (by invoking a local cloning method). Unauthorised access by clients is prohibited by requiring a clerk password. Clerks can also extend existing accounts with phone banking functionality. This is achieved by invoking a local imperative mixin method.

```
[
account local variable:
  [ clerk  local variable: "ClerkPwd";
    user   local variable: "UserPwd";
    amount local variable: 5000;
    deposit:val public imperative method:
      [ amount:amount+val ];
    withdraw:val userPwd:pwd public imperative method:
      [ (pwd=user) ifTrue: amount:amount-val ];
    newAccount:initval userPwd:upwd clerkPwd:cpwd public functional method:
      [ makeClone local cloning:
          [ user:   upwd;
            amount: initval
          ];
        (cpwd=clerk) ifTrue: makeClone return
      ];
    phoneBanking:cpwd numericCode:num public imperative method:
      [ makePhoneAccount local imperative mixin:
          [ code local variable: num;
            remoteWithdraw:val numericPwd:npwd public imperative method:
              [ (npwd=code) ifTrue: amount:amount-val ]
          ];
        (cpwd=clerk) ifTrue: makePhoneAccount
      ]
    ];

account deposit:1000;

"Change the account into a phone banking account with numeric code 2341" comment;
account phoneBanking:"ClerkPwd" numericCode:"2341";
account remoteWithdraw:500 numericPwd:"2341";

"Clone the account with initial amount 10000 and password NewUser" comment;
account2 variable: account newAccount:10000 userPwd:"NewUser" clerkPwd:"ClerkPwd";
account withdraw:2000 userPwd:"UserPwd"
]
```

3. Reintroducing safety in prototype-based languages

Agora objects can only be extended by sending them a message that invokes a mixin-method. This kind of inheritance was baptised *encapsulated inheritance on objects* because the potential extensions of an object are encapsulated within that object[1]. Encapsulated inheritance on objects solves the encapsulation problem that is inherent to all prototype-based languages featuring dynamic object extension [Steyaert&DeMeuter95] [Dony&al.92]. Indeed, due to dynamic object extension it is easy to breach object encapsulation, since objects can always be inherited from and inheritance is (due to late binding of self) by definition an encapsulation breaching mechanism [Snyder87].

Like object extension, cloning is accomplished by message sending, making it impossible to clone an object 'from the outside'. The object itself decides how it can be cloned by implementing the necessary cloning methods. For this reason we speak of *encapsulated cloning*. Encapsulated cloning allows to avoid the so called prototype-corruption problem [Blaschek94], which arises when users of prototypes accidentally change the internal state of a prototype instead of a clone of the prototype. One possible solution is to preclude prototypes from being changed. This is exactly what happens in class-based languages, where a distinction is made between unchangeable entities (classes) and changeable entities (objects). The alternative solution to the prototype corruption problem is to control the way copies of prototypes are created, and this is precisely what cloning methods do.

As will be discussed below, both mixin-methods and cloning methods are consequences of the very secure Agora meta object protocol.

---

[1]The theoretical difference between encapsulated inheritance on objects and other inheritance mechanisms is discussed in [Steyaert&DeMeuter95] and [Mens&al.96].

4. The Agora Meta Object Protocol

Agora is a reflective language which means that it is possible to influence the implementation of Agora from within Agora. New reifiers can thus be implemented in Agora itself. Agora's reflective architecture (described in [Steyaert94]) is based upon a linguistic symbiosis mechanism between Agora and its object-oriented implementation language, which means that Agora objects and implementation level objects can be freely intermixed. Thanks to this symbiosis, the implementation structures of the interpreter can be adapted or replaced by structures written in Agora itself.

The implementation level representation of an Agora object is called its *meta object*. The set of messages understood by meta objects is called the *meta object protocol* (MOP). The Agora MOP consists solely of a *send* message. This reflects the fact that Agora objects only understand messages. The following code fragment shows what the MOP looks like in a C++-like language:

```
class Object
     { private:
          ...
       public:
          Object send(Message m, ListOfObjects args);
     };
```

When writing meta programs (i.e. Agora code that explicitly deals with meta objects), the implementation level objects that become visible in Agora always satisfy the above protocol. Thus, it is impossible to bypass the message sending paradigm by 'going meta'. The only meta operation applicable to objects is message sending[2].

The inheritance and cloning mechanisms of Agora are direct consequences of this simple MOP: every Agora object *knows itself unencapsulated* (the private information in the above code fragment) but can only be accessed by the 'send' operation. Once a message arrives in an object, the object knows about its internal structures and can use these structures to deliver a clone or an extension of itself.

5. Conclusion and Future Work

The Agora approach differs fundamentally from other prototype-based languages, where any object can be cloned or extended by any client through explicit cloning and extension operators. In Agora the responsibility of extending or cloning an object is a contract between the object under consideration and the client requesting the extension or clone.

The very simple meta object protocol lying at the heart of the Agora philosophy only contains a message sending operation. This gives rise to very controlled extension, cloning and reflection mechanisms. The degree to which these operations are controlled by Agora determines the safety of Agora programs. A programmer is precluded from making accidental extensions, and although prototypes can be changed, the prototype corruption problem can easily be avoided using cloning methods. Furthermore, the safety barriers cannot be broken by using the reflective facilities of Agora. It is our strongest belief that this kind of safety is a necessary condition for prototype-based programming languages to become ready for the prime time.

An essential ingredient of safe programming languages is static typing. A static type system for dynamic extensions is being designed at our lab [Lucas&al.95], and remains to be fit into Agora. Another important field of future study is to find out how Agora (and prototype-based languages in general) handles the current shift in emphasis from 'plain object-oriented programming' to more abstract software engineering notions like frameworks, design reuse and contracts. In this context we plan to incorporate the work of [Steyaert&al.96] in Agora. The design reuse formalism that is introduced by this work continues the "safety philosophy" that is akin to Agora.

---

[2]This is very similar to functional languages whose only meta operation is 'apply'.

References

[Blaschek94] Blaschek, G. - 1994. Object-Oriented Programming with Prototypes; Springer-Verlag.

[Codenie&al.94] Codenie, W.; De Hondt, K.; D'Hondt, T. & Steyaert, P. - 1994. Agora: Message Passing as a Foundation for Exploring OO Language Concepts; SIGPLAN Notices, Volume 29, Number 12, December 1994, pp. 48-58, ACM Press.

[DeMeuter96] De Meuter, W. - 1996. Agora 96 User Manual; Programming Technology Lab, Vrije Universiteit Brussel.

[Dony&al.92] Dony, C.; Malenfant, J. & Cointe, P. - 1992. Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation; OOPSLA '92 Proceedings, pp. 201-217, ACM Press.

[Lucas&al.95] Lucas, C.; Mens, K. & Steyaert, P. - 1995. Typing Dynamic Inheritance: A Trade-off between Substitutability and Extensibility; Technical Report vub-prog-tr-95-03, Vrije Universiteit Brussel.

[Mens&al.96] Mens, K.; De Volder, K. & Mens, T. - 1996. A Layered Calculus for Encapsulated Object Modification; Submitted to Foundations of Object-Oriented Languages Workshop 3.

[Smith82] Smith, B. C. - 1982. Procedural Reflection in Programming Languages; PhD thesis, MIT.

[Snyder87] Snyder, A. - 1987. Inheritance and the Development of Encapsulated Software Components; Research Directions in Object-Oriented Programming; pp. 165-188, MIT Press.

[Steyaert94] Steyaert, P. - 1994. Open Design of Object-Oriented Languages, A Foundation for Specialisable Reflective Language Frameworks; PhD thesis, Vrije Universiteit Brussel.

[Steyaert&al.93] Steyaert, P.; Codenie, W.; D'Hondt, T.; De Hondt, K.; Lucas, C. & Van Limberghen, M. - 1993. Nested Mixin-Methods in Agora; ECOOP '93 Proceedings, LNCS 707, pp. 197-219, Springer-Verlag.

[Steyaert&al.96] Steyaert, P.; Lucas, C.; Mens, K. & D'Hondt, T. - 1996. Reuse Contracts: Managing the Evolution of Reusable Assets; To appear in OOPSLA '96 Proceedings, ACM Press.

[Steyaert&DeMeuter95] Steyaert, P. & De Meuter, W. - 1995. A Marriage of Class- and Object-Based Inheritance Without Unwanted Children; ECOOP '95 Proceedings, LNCS 952, pp. 127-144, Springer-Verlag.

[Taivalsaari93] Taivalsaari, A. - 1993. A Critical View of Inheritance and Reusability in Object-oriented Programming; PhD thesis, University of Jyväskylä.

[Ungar&Smith87] Ungar, D. & Smith, R. B. - 1987. Self: The Power of Simplicity; OOPSLA '87 Proceedings, pp. 227-242, ACM Sigplan Notices, Vol. 22, No. 12, ACM Press.