Agora:
The Story of the Simplest MOP in the World
- or -
The Scheme of Object-Orientation

Wolfgang De Meuter (wdmeuter@vub.ac.be)
Programming Technology Lab
Department of Computer Science
Vrije Universiteit Brussel
Pleinlaan 2 - 1050 Brussels - Belgium

Abstract

This paper discusses the design and implementation of the Agora language. Many ideas behind Agora come from Scheme, to our opinion, still one of the pearls of programming languages.

The most prominent ideas of Scheme are 1) Everything is a first class value, 2) Scheme programs look the same as Scheme data structures, 3) Function application and special form application are the only control structures, and 4) The language can be extended by writing new special forms.

One of the contributions of Agora is to transform these features into object-orientation. 1) In Agora, everything is an object. 2) Agora programs themselves are nothing but objects, the data structures of Agora. 3) The only control structures of Agora are messages and 'reifier messages', the object-oriented analogue of special forms. 4) Agora can be easily extended by writing new reifier messages.

We will show that all these Agora characteristics are an immediate consequence of its extremely simple meta-object protocol (MOP).

1 Introduction

This paper reports on several years of research on the Agora project. Agora is a pure object-oriented programming language (OOPL) which means that objects are its only language values and message passing is its only control structure. This is quite exceptional as OOPLs either allow a wide range of language values such as primitive values, objects, classes and interfaces, or, they only feature objects but then define many operations on them such as message passing, parent assignment, adding a slot to an object, cloning and so on. When we started the Agora project, we felt that such constructions were often chosen ad-hoc and have no general 'theory' behind them. That is why the Agora project was started with a minimal OOPL in which nothing but objects and messages where incorporated. We planned to add richer features on top of this model in

order to study the semantic relationships between these features and the basic message passing model. To our own surprise however, it was perfectly possible to add these features to Agora without having to change the initial model at all. Agora showed us that it *is* possible to construct a layered language in which the bottom layer solely consists of objects and messages, and onto which new layers defining inheritance, cloning and reflection can be added independently, without having to change the original notions of objects and message passing. Although this may sound like a trivial outcome, we will show that it is not.

Generally speaking, there are two ways of studying programming languages:

- The *pragmatic point of view* studies languages from a programmer's perspective. The main issue here is how a language can improve the way programmers think about their problems and structure their systems.
- The *language-theoretical point of view* studies programming languages from a conceptual angle. Issues such as orthogonality, semantic simplicity and regularity are the main concerns.

In conceiving Agora, we were strongly inspired by Scheme because we think Scheme is an excellent combination of both points of view. On the one hand, Scheme is conceptually well-understood and an easy-to-read formal description of it exist [3]. On the other hand, Scheme is designed with sufficient pragmatism in mind such that it can be used to write realistic software.

Several papers on Agora have already been published. These vary from object-oriented calculi [11, 12, 10] and denotational semantics [15] that underly Agora, towards more pragmatic explanations of how its constructions can be used to structure programs [8, 14, 4]. In this paper, we follow the approach taken in [13] and try to explain the big picture behind Agora, much in the spirit of the famous Abelson and Sussman course [1] that explains the entire Scheme story in one book. However the paper is not just a reformulation of other papers, but explains material that has never been published before, such as the reflective architecture of Agora. Still, the paper contracts the ideas of earlier Agora publications in one general philosophy.

2 Research Criteria

In order to conduct our search for a minimal OOPL, we used a number of criteria that naturally influenced the outcome of the research:

- Although OOPLs come in several variants [19], the main thing they have in common are objects and messages. While classes and inheritance are often considered inherent to object-orientation, they merely give us the ability to group several objects together. Hence we started with the hypothesis of making a *prototype-based language* [17, 16, 2].
- We chose to design a *reflective language*. One of the reasons for making Agora reflective was to enforce a well-designed semantics for it. Indeed, by making a language reflective, it becomes possible to alter its interpreter from within itself. Thus, one has to reify[1] the implementation details into

[1] To reify means 'to become a thing', 'to materialize'.

the language, and so the properties of the interpreter become very important because they eventually become visible to the programmer. After implementing Agora several times[2], we still think that adding reflection operators to a language is a good research methodology to enforce its simplicity, orthogonality and consistency.

- We used *the modular interpreters approach* [7], in which programming languages are constructed starting with an empty language and adding features one by one. Adding new features often requires additional semantic structures which sometimes require a complete redesign of the existing semantics because they interact with the existing ones in a non-orthogonal way. The modular interpreters approach favors a layered way of designing languages such that the interactions between the different layers can be easily detected and studied. We used the same methodology: our goal was to start with nothing but objects and messages and to investigate how adding other features influenced this basic paradigm.

- The final hypothesis is that, in resolving the design alternatives for Agora, we tried *to stick as close to Scheme as possible*. Hence, Agora can be regarded as the Scheme of object-orientation. We could not follow Scheme when designing Agora's reflective facilities, because Scheme is not reflective. In these cases, the object-oriented transposition of Brown [18] was chosen. Brown is a fully reflective 'closure' of Scheme.

3 The Agora Language

This section introduces Agora without taking reflection into account. In this vanilla variant, the emphasis is completely on objects and messages. Section 3.1 explains Agora messages, sections 3.2, 3.3 and 3.4 show the nature of objects.

3.1 Agora Message Categories

Scheme distinguishes between 'ordinary functions' and 'special forms'. Whereas Scheme normally uses applicative order evaluation for its 'ordinary' functions, special forms are evaluated in a different way. Consider for example the `(define x 2)` expression. When encountering it, Scheme will apply the 'define' procedure onto an unevaluated `x` parameter and an evaluated `2` parameter. The fact that `x` is not evaluated and that `2` is evaluated depends on `define`. Each special form uses its own evaluation order.

Likewise, Agora knows two kinds of messages: ordinary messages and reifier messages. Ordinary messages correspond to ordinary function applications. An example is `3 + 4` where `+` is sent to the evaluated `3` with the evaluated `4` as argument. Reifier messages correspond to the special forms of Scheme. An example is `x VARIABLE:3` which is used to define a variable in Agora by sending the `VARIABLE:` message to the identifier `x` with the expression `3` as parameter. Hence, as in Scheme, the essential difference between ordinary messages and reifier messages is their evaluation order. Agora reifier messages always consist of completely capitalized identifiers.

[2] So far, Agora has been implemented in Scheme, Smalltalk, C++ and Java.

Besides the distiction between ordinary and reifier message expressions, Agora distinguishes between receiverful and receiverless message expressions. Receiverless message expressions are exactly the same as 'receiverful' message expressions except that their receiver is syntactically missing. If we wouldn't have them, it would be impossible to write realistic programs because the only way to write programs using messages and objects is to write down messages to messages to ... to messages to objects. Although orthogonal and simple, this is too much λ-calculus like to be of any use for writing real programs: it would not even be possible to declare a variable since the name of the variable is not an object, nor a message. At this point in the paper, the meaning of receiverless message expressions is not yet important but it is not wrong to think of them as function calls.

As a final note on Agora's syntax we mention that all messages come in unary, operator and keyword form, just like in Smalltalk. Hence, Agora features 12 kinds of messages that vary along the dimensions $\{receiverless, receiverful\}$, $\{ordinary, reifier\}$ and $\{unary, operator, keyword\}$. The complete system of messages is summarized in figure 1. We do not give an example of operator reifier messages, since there is no capitatised analogue of things like `+`. Until now we did not need them in our text-based variants of Agora. In a decent Agora environment (like the Smalltalk implementation), reifiers are denoted boldfaced instead of capitalized, and then reifier operators are perfectly possible[3].

Receiverful		Operator	`3 + 4`
	Ordinary	Unary	`3 abs`
		Keyword	`dict at:"key"`
		Operator	
	Reifier	Unary	`message SUPER`
		Keyword	`myVariable VARIABLE: 4`
Receiverless		Operator	`- 5`
	Ordinary	Unary	`myVariable`
		Keyword	`at:key put:athing`
		Operator	
	Reifier	Unary	`SELF`
		Keyword	`CURRENTLY:not IN:use`

Figure 1: Agora Syntactic Message Categories

Example: The following expression elaborates on the Agora syntax. The informal meaning of the expression is to install a method `compute:value` in the object in which the expression occurs.

```
compute:value METHOD: ((SELF try:value) + (5 abs))
```

This is a reifier keyword message `METHOD:`. Its receiver is the receiverless ordinary keyword message expression `compute:value`, here acting as a formal pattern. The argument `value` is a receiverless ordinary unary message expression. The argument of the `METHOD:` message is an ordinary operator message

[3] Note that this is just a lexical problem that does not change the fundamental ideas of Agora.

expression `+` with `5 abs` as argument. The receiver of the operator message is an ordinary keyword message expression `SELF try:value` with `SELF` as receiver. `SELF` is a receiverless reifier unary message expression.

3.2 Agora Objects

The simplest way to create an Agora object is to denote it by means of a literal. Agora supports integer literals, float literals, character literals, string literals, boolean literals and `null`. The corresponding objects are native and the evaluator automatically creates them upon encountering a literal.

Besides these built-in objects, the Agora programmer can also create her own objects by simply writing them down, similar to the way Scheme programmers can make a new function by simply writing it down using the `lambda` special form. In Agora, such an 'ex nihilo' created object is constructed by listing the slots of the object between square brackets, separated by semicolons.

```
[ ...; ...; ...;    ... ]
```

The entries of the object must be valid Agora message expressions. This includes ordinary and reifier messages. Using reifier message expressions like `x VARIABLE:4`, the programmer can install attributes in the object.

3.3 Basic Agora Reifiers

A particular variant of Scheme is defined by the implemented special forms. This property is often used (using `define-macro`) in programming language courses, where Scheme is enriched with new constructions. Likewise, variants of Agora are defined by the implemented reifier messages. Therefore, Agora is a language family instead of a language. This section explains the elementary reifier messages defined in most members of the family[4]. Most of these reifiers install new attributes in the slots of ex nihilo created objects.

3.3.1 Variables

The simplest kind of attributes one can install in an object's slots are variables. Variables are created by sending the `VARIABLE:` reifier message to an identifier with the initial value of the variable as argument. The `VARIABLE:` message installs two slots. Whenever a variable `x` is declared by sending the message `x VARIABLE: 5`, a reading slot named `x` and a writing slot named `x:` are installed. These slots are accessor methods to the variable. As in Self, users of the object can read the variable by sending `x` to the object. They can modify the variable by sending `x:` with the new value as an argument.

3.3.2 Methods

As illustrated before, sending the `METHOD:` reifier message installs a method in an object. The receiver of that message must be a receiverless message expression acting as the formal pattern of the new method. The argument of `METHOD:` can be any expression serving as the body of the method. If more than one

[4] The exact technicalities of the reifiers discussed here come from Agora98, the most recent instantiation of Agora in Java [5].

expression determines the body, they may be grouped between curly braces (as in Java) and separated by semicolons.

Following the object-oriented tradition, recursive methods are programmed by sending messages to `SELF`. Evaluation of this receiverless reifier unary message expression always returns the 'current' receiver.

3.3.3 Cloning Methods

Agora does not support a primitive cloning operator because we did not want to augment our message passing language by additional built-in operators. Instead, a specific kind of attribute, called a cloning method, must be used to clone objects. A cloning method is installed by sending the message `CLONING:` to a pattern, just like an ordinary method is declared. Upon invocation of an installed cloning method, its body is executed in the context of a clone of the receiver instead of the context of the receiver itself. The following example illustrates this. In the example, a `new` cloning method is installed in the ex nihilo created `listnode` object. Upon sending `new` to the `listnode`, the `next` and `elmt` variables in the copy are initialised to `null`. By default, the result of a cloning method is the copy of the receiver to which the 'cloning message' was sent. Note that the evaluation of `SELF` in a cloning method returns the copy of the receiver.

```
listnode VARIABLE:
   [ next VARIABLE: null;
     elmt VARIABLE: null;
     new CLONING:
      { SELF next:null;
        SELF elmt:null
      }
   ]
```

3.3.4 Views

Agora does not feature built-in operators to add or delete slots to and from objects. Instead, a special kind of attribute, called a view, can be installed in an object[5]. A view is a method whose body contains a group (between { and }) of expressions which will be evaluated in a new object that has the receiver as parent link. The following expression gives an example of a point onto which circle views can be laid by sending `circle:` messages to the `point` object.

```
point VARIABLE:
  [ x VARIABLE:0;
    y VARIABLE:0;
    circle:r VIEW:
      { radius VARIABLE:r;
        inCircle:p METHOD:
          { ((p x) sqr + (p y) sqr) sqrt <= (SELF radius) }
      }
  ]
```

[5] These views are also called 'functional mixin-methods' in previous versions of Agora.

A view does not destructively change the receiving object. When a `circle:` message is sent to the `point`, a new object is created with the receiver (`point`) as parent-of link. The slots of this extension are determined by evaluating the body of the view in the context of the extension. The extension is the result of sending the 'view message'.

3.3.5 Mixins

While views do not destructively change their receiver, mixins do[6]. In the following example, sending `circle:` to the `point` object, really adds a `radius` variable and an `inCircle:` method to the original point. All objects that can access the point, can now also access `radius` and `inCircle:`.

```
point VARIABLE:
  [ x VARIABLE:0;
    y VARIABLE:0;
    circle:r MIXIN:
      { radius VARIABLE:0;
        inCircle:p METHOD:
          { ((p x) sqr + (p y) sqr) sqrt <= (SELF radius) }
       }
  ]
```

It is important to understand the difference between views and mixins. While views only put an extra inheriting layer around an object, with the original object as parent, mixins really change the object. Everyone refering to the object, including views, will notice that the object has been destructively extended. Mixins thus allow one to change an entire object hierarchy in one stroke.

It is important to notice from sections 3.3.1, 3.3.2, 3.3.3, 3.3.4 and 3.3.5 that Agora is a full-fledged OOPL with inheritance and cloning. Nevertheless, objects are its only language values and message passing is its only built-in operation.

3.3.6 Other Reifiers

Although many reifiers exist, figure 2 gives an overview and a short description of a few frequently used ones. See the language manual [5] for more details.

`SUPER`	Forwards a message to the parent object.
`TRY:CATCH:`	Tries an expression and catches an exception when needed.
`RAISE`	Raises an exception.
`IFTRUE:IFFALSE:`	Tests a conditional and evaluates one of the branches.
`WHILETRUE:`	Leading-condition loop.
`UNTILTRUE:`	Trailing-condition loop.
`FOR:TO:DO:`	Bounded loop.

Figure 2: Frequently Used Reifiers

[6] In previous papers, these mixins where termed 'imperative mixin-methods' as they really change the receiving object.

3.4 Local and Public Attributes

Up until now we have explained what ordinary and reifier messages are good for. The semantics of receiverless messages has not yet been explained.

Agora objects actually consist of a local part and a public part. The public part can be accessed by anyone. The local part can only be accessed by the object itself, and this is what receiverless messages are good for: receiverless messages are messages to the local part of the object in which they occur. Thus, when encountering a message `msg:arg` there are two possibilities. If the message is receiverless, then there must be a message `msg:` in the local part of the object. The other possibility is that it concerns a receiverful message. Then, there must be a message `msg:` in the public part of the receiver of the message. Inside an object, this receiver will be `SELF`. As such, an object can send two kinds of messages to itself. Receiverless messages are sent to its local part, while receiverful messages to `SELF` are sent to its public part. `SELF` is a receiverless unary reifier message expression that always returns the current receiver.

Declaring an attribute using `VARIABLE:`, `METHOD:`, `CLONING:`, `VIEW:` or `MIXIN:`, by default installs that attribute in the public part of the object in which the declaring expression occurs. In order to define an attribute locally, the unary message `LOCAL` can be used. In the following example, the `elmt` variable is accessible to everyone while the `next` variable is only visible to methods declared inside the list node.

```
listnode VARIABLE:
   [ elmt VARIABLE: null;  // same as elmt PUBLIC VARIABLE:null
     next LOCAL VARIABLE: null
   ]
```

The default modifying unary message is `PUBLIC`. One can also use both modifiers such as in `id:x PUBLIC LOCAL METHOD: x`. In this case, the `id:` method is accessible from the outside. From the inside, it is accessible by `SELF id:4` because it is public, as well as by `id:4` because it is local.

4 Evaluating Agora Expressions

In order to completely understand Agora, it is necessary to have a look at its semantics. Again the parallel with Scheme will be drawn.

4.1 General

Looking at the internal details of a Scheme evaluator [1], it essentially consist of the following ingredients:

- A memory of cons cells containing Scheme data structures and programs. Both are internally represented as Scheme lists.
- A procedure *eval* that can be applied to any Scheme list. *eval* dispatches over its argument list and calls the appropriate evaluation rule for it.
- An environment system binding names to their values. *eval* is parameterised by an environment parameter with which it evaluates the expression at hand. The environment is recursively passed down the evaluator.

- A procedure *apply* to apply a function onto a suite of arguments. When this happens, the body of the function is evaluated with *eval* which might again consist of function applications that are handled by *apply*. This recursive game between *eval* and *apply* ends when basic syntax like numbers is evaluated.

An Agora evaluator is implemented in an object-oriented language[7] and this is part of the definition of Agora, just like functional list processing (i.e. applying procedures on lists) is an inherent part of a Scheme evaluator. But besides this difference, each of the above ingredients can be found back in Agora, albeit translated into its object-oriented equivalent:

- A memory of objects. These objects represent data structures (i.e. Agora objects) and parse trees (i.e. Agora programs). Hence, just as in Scheme, programs and data structures are represented in the same way.

- All Agora parse tree objects understand the message *eval*. While the Scheme *eval* dispatches over the expressions it has to evaluate, this dispatching is automatic in the Agora implementation: *eval* is sent to a parse tree such that the message automatically arrives at the right implementation.

- The *eval* message is parameterised by a context object that represents the environment in which the expression is evaluated. This context object recursively travels through the evaluator. It contains a reference to the 'current' lexical scope, the 'current' self, the 'current' parent etc.

- Each Agora object is internally represented by an implementation level object. This implementation level object understands a message *send* which takes a message and a list of arguments and produces another Agora object. We consider *send* as the object-oriented analogue of *apply*.

In the same way that the execution of a Scheme program can be considered as a recursive game between *eval* and *apply*, the execution of Agora programs can be seen as an alternating interaction of *eval* and *send*. This is a very fundamental notion which forms the basis for all the language design decisions that have been made for Agora. The following properties further elaborate on the fundamentality of *send* and *eval*:

- Seen through the eyes of the Scheme evaluator, functions are represented as an abstract data type onto which *apply* is the *only* operation. Once *apply* is called on a function, it can access the internal details of the function consisting of its formal parameters, its lexical environment and its body code. However, these constituents are invisible outside the function concept. It would not be a good idea to define operations like *getLexicalEnv* and *getFormals* and *getBody* on functions, since this would reveal the internal representation of functions[8]. Therefore, Scheme only defines *apply* on functions. In the same way, seen through the eyes

[7] Notice that we have also created an evaluator in Scheme, but still, closures were used to implement the implementation structures. Closures can be regarded as objects.

[8] In denotational semantics terminology, one says that the evaluator must be fully abstract.

of the Agora evaluator, *send* is the only message on Agora objects. Defining more meta messages on objects would reveal more information about objects than necessary [15].

- In Scheme, *apply* takes a function and a list of arguments. *apply* evaluates the body of the function using the environment of definition, augmented with new bindings of arguments to formal parameters. The same is true for the *send* message in Agora. *send* takes the name of the message to be sent together with a list of actual arguments. It uses *no* hidden arguments like self references, environments or whatsoever, but evaluates the method corresponding to the message in the context of the receiving object and the actual arguments.

The remainder of section 4 will discuss these notions in depth. Section 4.2 summarizes the abstract syntax of Agora. Section 4.3 elaborates on the way Agora objects are constructed and what *send* looks like in detail. Finally, sections 4.4 and 4.5 focus on *eval*.

4.2 Agora Abstract Syntax

Figure 3 gives an overview of the Agora *abstract* syntax. It consists of basic literals, ex-nihilo created objects, groups of expressions in methods, ordinary messages, reifier messages and their receiverless equivalents. The difference between operator, unary and keyword messages exists only at a lexical level and does not influence Agora's abstract syntax because a unary message is nothing but a message with zero parameters and an operator message is just a message whose pattern is made up of special operator symbols.

Basic Literals	b
Ex-nihilo Objects	$[e_1, \ldots, e_n]$
Grouped Expressions	$\{e_1, \ldots, e_n\}$
Ordinary Message Expressions	$e.m(e_1, \ldots, e_n)$
Reifier Message Expressions	$e.M(e_1, \ldots, e_n)$
Ordinary Receiverless Message	$m(e_1, \ldots, e_n)$
Reifier Receiverless Message	$M(e_1, \ldots, e_n)$

Figure 3: Agora Abstract Syntax

4.3 Object Structures

As already explained, an Agora object consists of a local part and a public part. Internally, these are tied together by an object identity. Hence, every ex nihilo created object (a 'self') has a reference to an object identity. The object identity has a reference to the public part and the local part of the self (i.e. the object). An example of such a typical object structure can be found in figure 4. The reason for making a distinction between objects and their identities will be given in section 4.5.
The self is the only part of the object users are allowed to see. It is the entity on which the *send* message is implemented. Internally *send* is implemented by

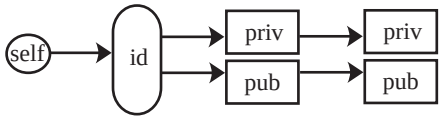

Figure 4: Agora Object's Structure

forwarding the message through the chains of parts the object contains from successive applications of views and mixins. Once the attribute corresponding to the message is found, it is processed. While forwarding messages through the object structure, a context object is built that contains the internal parts of the object. This context is necessary to evaluate a method. Indeed, we saw in section 4.1 that *eval* is parameterised by a context object. This context consists of the information of the object in which the method to be evaluated occurs.

Now we are ready to explain the difference between message passing and delegation in Agora. Message passing is used to send messages to objects using the *send* operation that is only parameterised by the message pattern and the arguments. Delegation in Agora consists of implicitely traversing the different parts of the object structure (following parent-of links formed by succesive invocation of view and/or mixin attributes) until the method is found. The distinguishing thing between message passing and delegation is the context object. In an object, nothing is allowed to enter but the message pattern and the arguments. Delegation on the other hand, juggles around lots of extra 'hidden' arguments in a context object [9].

In order to stay consistent with the terminology used in [15], the internal parts of the object (the identity, the public and local parts) will henceforth be called *generators*. Hence, objects understand the message *send* while generators implement the message *delegate*. Apart from the message pattern and the parameters, *delegate* is parameterised by a context object containing the internal parts of the object that received the message through *send*.

The Agora message passing operator is summarised in figure 5. When a message m is sent to an object o with arguments $a_1, \ldots, a_n$, o refers to its own identity generator (notice that *id* is only visible for the object). From this identity, it references the public part generator and delegates the message to it. In each generator g, delegating the message consist of looking up the message in the method table. If it is there, the corresponding attribute is invoked by sending it *do*. If it is not there, the message is delegated to the following generator in the chain. In this delegation process, a context object c is passed around that contains references to the object itself, the identity generator, the public generator and the local generator. This context is used in *do* in order to evaluate the body code of the found attribute. Note that all the semantic rules in this paper are given in an imaginary OOPL. As we have said before, the fact that Agora is implemented in an OO medium is inherent to the definition of Agora. We will explain this further in section 5.

$$o.send(m, a_1, \ldots, a_n) = o.id.pub.delegate(m, c, a_1, \ldots, a_n)$$

$$\text{where } c = \begin{pmatrix} \text{self} & = & \text{o} \\ \text{loc} & = & \text{o.id.loc} \\ \text{pub} & = & \text{o.id.pub} \\ \text{id} & = & \text{o.id} \\ \text{parent} & = & \text{o.id.pub.parent} \end{pmatrix}$$

$$g.delegate(m, c, a_1, \ldots, a_n) = \begin{cases} att.do(c[parent \rightarrow g.parent], a_1, \ldots, a_n) & \text{if } att = g.lookup(m) \\ g.parent.delegate(m, c, a_1, \ldots, a_n) & \text{otherwise} \end{cases}$$

Figure 5: Agora Message Passing Operator

4.4 Evaluation Rules

Let us now turn to the evaluation rules for the syntax outlined in figure 3. These rules are given in figure 6, in which we use angular brackets $\langle$ and $\rangle$ to surround parse tree objects.

$\langle literal \rangle.eval(c)$	Create new basic object understanding *send*
$\langle [e_1; \ldots; e_n] \rangle.eval(c)$	$\langle e_1 \rangle.eval(c'); \ldots; \langle e_n \rangle.eval(c')$ where $c' = \begin{pmatrix} loc & = & c.loc.addFrame() \\ pub & = & rootPublic.addFrame() \\ id & = & \text{new Identity}(pub, loc) \\ parent & = & rootPublic \\ self & = & \text{new Object}(id) \end{pmatrix}$
$\langle \{e_1; \ldots; e_n\} \rangle.eval(c)$	$\langle e_1 \rangle.eval(c); \ldots; \langle e_n \rangle.eval(c)$
$\langle e.m(e_1, \ldots, e_n) \rangle.eval(c)$	$\langle e \rangle.eval(c).send(m, \langle e_1 \rangle.eval(c), \ldots, \langle e_n \rangle.eval(c))$
$\langle m(e_1, \ldots, e_n) \rangle.eval(c)$	$c.loc.delegate(m, c, \langle e_1 \rangle.eval(c), \ldots, \langle e_n \rangle.eval(c))$
$\langle e.M(e_1, \ldots, e_n) \rangle.eval(c)$	$adHocEval(c, \langle e \rangle, M, \langle e_1 \rangle, \ldots, \langle e_n \rangle)$
$\langle M(e_1, \ldots, e_n) \rangle.eval(c)$	$adHocEval(c, M, \langle e_1 \rangle, \ldots, \langle e_n \rangle)$

Figure 6: Evaluation Rules (part 1)

- The evaluation rule for basic literals consists of creating a new basic object which is also supposed to understand *send* as its only implementation level message.
- Ex nihilo created objects are constructed by creating new generators for the public and the local parts. From the semantics we see that this is accomplished by attaching a frame (i.e. a generator with a parent-of link) to the public of the unique Agora root object such that ex nihilo created objects understand all the messages of the root. The local generator of the newly created object consists of a frame attached to the local generator of the context (i.e. the local generator of the object in which the `[...]`

expression occurs). Hence, the local part of objects is lexically scoped: an object can access the attributes from the local part of the context in which it occurs.

- Evaluating a group of expressions in a context simply consists of evaluating all the expressions from left to right in that context.
- Evaluating an ordinary message $e.m(e_1, \ldots, e_n)$ is accomplished by evaluating the receiver and the actual arguments in the same context, and then sending the message using *send*.
- As explained in section 3.4, receiverless messages are looked up in the local generator of the object. This local generator can be read from the evaluation context, since it was put there when *send* was invoked.
- The final syntactic category to be evaluated are reifier messages and receiverless reifier messages. As explained, reifier messages are the object-oriented analogue of special forms. They are thus handled in an ad-hoc manner. Let us elaborate on this by looking at the `VARIABLE:` reifier message. If this reifier message is sent (e.g. `x VARIABLE:3`), the ad-hoc evaluation strategy will evaluate the initial value in the given context, and will install a read and a write slot in the public generator that resides in the context. Likewise, evaluating `circle LOCAL VIEW: ...` gives rise to a view slot being installed in the local generator of the object in which the expression occurs, that is, the local generator of the context.

4.5 The power of attributes

As we have seen in figure 5, a message to an Agora object is sent by looking it up using *delegate*. Once the message has been found, $do(c, a_1, \ldots, a_n)$ is sent to the corresponding attribute. This attribute can be a variable accessor slot, a method, a view, a mixin or a cloning method. This is pretty much the power of the Agora paradigm: instead of defining all kinds of operators on objects such as cloning, slot addition and deletion, parent assignment and so on, attributes of an object are evaluated in the context c containing the internal details of the object in which the attribute was found. Each kind of attribute knows what to *do* with these details. Stated otherwise, Agora objects are strongly encapsulated (due to *send*) but upon invocation of a message, attributes can access the internal generators of the object they reside in [15]. Based on these internal generators, extensions and clones of the receiver can be returned from the attributes.

Let us briefly go through figure 7 which gives the semantics for each kind of attribute discussed in section 3.3. Apart from the variable accessor methods, the technique used is always the same. When *do* is sent to an attribute object with a given context c, a new context c' is constructed to evaluate the body expression of the attribute. In each case, the local generator is extended to contain the formals-actuals bindings.

In the *do* for cloning methods, a copy of all the generators is created and a new object is made with the copies. The body of the method is evaluated in the context of the copied generators.

Particularly interesting is the difference between mixins and views. In both cases, generators are extended. In the case of views, a new object is created

$variableGet.do(c)$	return contents of the variable
$variableSet.do(c, a_1)$	assign variable to a_1
$method.do(c, a_1, \ldots, a_n)$	$method.bodyCode.eval(c')$ with $c' = \begin{pmatrix} loc & = & c.loc.addFrame(a_1, \ldots, a_n) \\ pub & = & c.pub \\ id & = & c.id \\ parent & = & c.parent \\ self & = & c.self \end{pmatrix}$
$cloning.do(c, a_1, \ldots, a_n)$	cloning.bodyCode.eval(c') with $c' = \begin{pmatrix} loc & = & c.loc.copy().addFrame(a_1, \ldots, a_n) \\ pub & = & c.pub.copy() \\ id & = & \text{new Identity}(pub, loc) \\ parent & = & c.parent.copy() \\ self & = & \text{new Object}(id) \end{pmatrix}$
$view.do(c, a_1, \ldots, a_n)$	view.bodyCode.eval(c') with $c' = \begin{pmatrix} loc & = & c.loc.addFrame(a_1, \ldots, a_n) \\ pub & = & c.id.addFrame() \\ id & = & \text{new Identity}(pub, loc) \\ parent & = & c.id \\ self & = & \text{new Object}(id) \end{pmatrix}$
$mixin.do(c, a_1, \ldots, a_n)$	mixin.bodyCode.eval(c') with $c' = \begin{pmatrix} loc & = & c.loc.addFrame(a_1, \ldots, a_n) \\ pub & = & c.pub.addFrame() \\ id & = & c.id.assign(pub, loc) \\ parent & = & c.pub \\ self & = & c.self \end{pmatrix}$

Figure 7: Agora Attribute Invocation

in which the body expression is evaluated. A view frame is attached to the *id* such that later mixins on the parent will affect that view. In the case of a mixin, the given object identity is provided with the extended public and local generator, but the object and the object identity stay the same. Thus, the difference between views and mixins is the main motivation for making a distinction between objects and object identities.

4.6 Discussion

In Agora, objects are strongly encapsulated and only understand *send*. Nevertheless, by cleverly using their internal structure, special purpose attributes enable extension and cloning. The more complex the internal structure of objects, the more information contexts carry around and thus the more kinds of attributes we can handle. This is the way we added inheritance and cloning to Agora, without changing the basic paradigm of objects and messages. To the best of our knowledge, Agora is the only language in which this is possible. Other prototype-based languages add these features by enriching their MOP with several operators such as cloning and slot addition and deletion. In [15], this was called a 'change in object model'. Seen from the modular interpreters

approach, Agora is a layered language: a new layer can be added to the language by making objects and contexts richer and adding extra attributes that use the additional information in their specific implementation of *do*. But no matter how rich the internal structure of objects might become, the evaluator only uses *send* such that the basic object model always stays the same.

A common criticism against Agora is that, although it is a nice theoretical model, it does not work in practice because all possible cloning methods, view and mixin methods are installed *inside* the object upon creation of the object. As such, it is not possible to reuse objects written by other people. For cloning, this is not a big problem as one can always install a cloning method `new CLONING:{}` in the root of the Agora hierarchy such that all decendants at least understand `new`. Extending objects 'from the outside' is also possible, but this requires a few additional constructs that will be defined in the next section.

5 Reification and Absorption

This section extends Agora with reflection operators. Reflection can be modelled as a combination of reifying implementation level structures into Agora, and absorbing Agora objects back into the evaluator. This treatment of reflection is mainly due to Wand and Friedman [18] who investigated reflection in the context of Brown, a fully reflective variant of Scheme. In Agora, this research was transposed to object-orientation.

5.1 Reification of Primitives

In order to talk about reflection, it is necessary to distinguish two levels in the Agora semantics. The 'down' level is the level of the (object-oriented) implementation language such as Java, Smalltalk or C++. The 'up' level is the Agora level being evaluated by the 'down' level. Using more standard terminology, the 'down' level is the meta level and the 'up' level is the base level.

As can be seen in figure 6, Agora actually knows two kinds of Agora objects. The most obvious ones are ex nihilo created objects that consist of chained generators. But as can be seen in the evaluation rule for basic literals, Agora also has built-in objects that understand the *send* meta message. These objects are actually wrapped versions of their corresponding 'down' object. For example, the Agora literal `3` is represented by wrapping the corresponding (Java, Smalltalk or C++) object 3. This wrapping process is accomplished by sending the *up* message to the implementation level object. Hence, $\langle 3 \rangle.up()$ is an Agora object that understands *send*. The implementation of *send* for upped objects will map every message onto the corresponding message at the down level. This is accomplished by bringing both the receiver and the arguments to the down level. After actually sending the message, the resulting down level object is brought back to the up level by sending *up*. This augmentation of the message passing operator (figure 5) for upped primitives is shown in figure 8. The evaluator therefore knows two kinds of objects with the same *send* interface. However, due to strong encapsulation (i.e. only *send* is possible), the evaluator is not able to distuingish between these two kinds of Agora objects: only the objects know whether they are ex nihilo created or upped implementation level

objects. They use this knowledge to implement *send* appropriately (i.e. figure 5 or 8).

$$o^u.send(m, a_1, \ldots, a_n) = o^u.down().m(a_1.down(), \ldots, a_n.down()).up()$$

Figure 8: Agora Message Passing Operator for Upped Primitives o^u

5.2 The Evaluator Reconsidered

The idea of upping implementation level objects can not only be applied to primitive literal objects, but to *all* implementation level objects. In each Agora implementation, all implementation level objects understand the message *up*[9]. This can be used to give a much cleaner semantics to reifier messages. Instead of handling a reifier message in an ad-hoc manner, the model with *up* treats reifier messages as real Agora messages to upped syntax objects. Together with the treatment of literals as discussed in the previous section, this gives us figure 9 which is an improved version of the evaluation rules outlined in figure 6. As we can see in figure 9, reifier messages are no longer implemented in an ad-hoc fashion. Instead they are really sent to the reified (i.e. upped) versions of their syntactically appearing receiver. This is why messages like `VARIABLE:` were called reifier messages in the first place: they are reifications of the corresponding messages defined on the implementation level objects that represent parse tree nodes. The advantage of processing reifier messages this way is that by the late binding of reifier messages (i.e. they are really looked up in the upped object), we can install our own reifiers which turns Agora into a reflective language. This is also the explanation why Agora must be implemented in an OO medium.

$\langle literal \rangle.eval(c)$	$\langle literal \rangle.up()$
$\langle e.M(e_1, \ldots, e_n) \rangle.eval(c)$	$\langle e \rangle.up().send(M, c.up(), \langle e_1 \rangle.up(), \ldots, \langle e_n \rangle.up()).down()$
$\langle M(e_1, \ldots, e_n) \rangle.eval(c)$	$c.loc.delegate(M, c.up(), \langle e_1 \rangle.up(), \ldots, \langle e_n \rangle.up()).down()$

Figure 9: Evaluation Rules (part 2)

In figure 10, we illustrate the computational process induced by sending the reifier message `x VARIABLE:3`. The applied rules are the one for evaluating a reifier message (figure 9) and the one for mapping Agora messages on upped objects onto their corresponding implementation level message (figure 8). When encountering the message `x VARIABLE:3`, the `VARIABLE:` message is sent to the upped version of `x` with the upped version of `3`. Of course, the result must be brought back to the evaluator level by sending it *down*. Because of the message passing operator outlined in figure 8, this means that the `VARIABLE:` Agora message will be mapped onto the implementation level message *variable* defined on identifiers (or more precisely: receiverless unary patterns).

[9] In each Agora implementation, we had to come up with a different technical trick to make sure every implementation level object understands *up*.

$\langle x.VARIABLE(3)\rangle.eval(c)$
$= \langle x\rangle.up().send(VARIABLE, c.up(), \langle 3\rangle.up()).down()$
$= \langle x\rangle.up().down().variable(c.up().down(), \langle 3\rangle.up().down()).up().down()$
$= \langle x\rangle.variable(c, \langle 3\rangle).up().down()$
$=$ (install slots 'x' and 'x:' in $c.pub$ bound to $\langle 3\rangle.eval(c)).up().down()$
$=$ (return $\langle 3\rangle.eval(c)).up().down()$
$=$ (return $\langle 3\rangle.up()).up().down()$
$= \langle 3\rangle.up().up().down()$
$= \langle 3\rangle.up()$ (i.e. the Agora object 3)

Figure 10: Example of Evaluating `x VARIABLE:3`

One of the distuinguishing features of reifier messages is that they are dynamically scoped: the first argument of each reifier method is the context in which the reifier message occurs. This is in contrast to ordinary methods that are completely lexically scoped. Parameterising reifier messages with their context of invocation is not some 'dirty' trick but completely follows the spirit of special forms in Scheme. When evaluating the `(if ... ... ... )` special form, the expressions have to be evaluated in the context where the special form occurs, and not in the context of definition of the 'if' procedure.

5.3 Absorbing Ex Nihilo Objects

In the previous sections we have seen that any 'down' level object can be reified in Agora by sending *up* and that an upped object can always be brought back to the 'down' level by sending it *down*. The final step for full reflection is to extend the *down* mechanism for ex nihilo created objects as well. This is called absorption. Absorption allows one to down an ex nihilo created Agora object into the implementation, such that the implementation can send messages to it. Each Agora implementation uses its own technical trick to accomplish this, depending on the implementation language. In Java for example, we have to dynamically generate class files in order to wrap an Agora object in a Java object. Each message sent to this native Java object must be mapped onto the corresponding message in Agora: if $o^d.m(a)$ is sent in Java, and o^d is a downed Agora object, the message must be resent in Agora yielding $o^d.up().send(m, a.up()).down()$. Hence, when sending implementation level messages to downed Agora objects, the receiver and all arguments must be upped. Then the message must be sent to the Agora object using *send*. The resulting Agora object must be brought back to the implementation language using *down*. The technique is summarised in figure 11. It allows us to replace implementation level objects (i.e. objects of the evaluator) by our own objects written in Agora. We show how to use this technique in section 6.

$$o^d.m(a_1, \ldots, a_n) = o^d.up().send(m, a_1.up(), \ldots, a_n.up()).down()$$

Figure 11: Message Passing For Downed Agora Objects

Combining the message passing operators of figures 5, 8 and 11 yields the implementation of *up* and *down* as shown in figure 12. Upping a downed up level object o^u consists of returning the original up level object o^u. Upping a non downed object consists of creating a new up level Agora wrapper for it. The same mechanism is used for *down*. As already mentioned, each Agora implementation uses its own technical trick to implement these rules.

$$\forall\ o : o.up() = \begin{cases} \text{new } UpWrapper(o) & \text{if } o \text{ is not a downed one} \\ o^u & \text{if } o = o^u.down() \end{cases}$$

$$\forall\ o \text{ understanding } send : o.down() = \begin{cases} \text{new } DownWrapper(o) & \text{if } o \text{ is not a upped one} \\ o_d & \text{if } o = o_d.up() \end{cases}$$

Figure 12: The Implementation of Up and Down

5.4 The Agora MOP

The system of 'upping' and 'downing' objects implies that *all objects* in the implementation understand *up*, and that *all Agora objects* understand *send*, *up* and *down*. These mechanisms form the meta object protocol of Agora and can be regarded as the object-oriented analogue of the meta functions used in Brown [18]. In Brown, function and special form application are seen as the only control structures[10], and the operators $\vee$ and $\wedge$ are the conversion operations between the base and the meta level. In the same way, message passing and reifier message passing (with hidden context argument) are the only control structures in Agora. *up* and *down* are used to consistently switch objects between the base level and the meta level[11].

6 Using Reflection

The semantic mechanisms outlined in the previous section open up a new scale of Agora constructions such as extending objects from the outside and reflective programming.

6.1 Extension From the Outside

Every Scheme programmer knows quoting, a mechanism to transform a program into a data structure. In Agora, quoting an expression is accomplished by sending it the `QUOTE` reifier. The result thereof is the expression itself in the

[10] In Brown, special forms are also called reifier functions.

[11] Note that implementing *down* directly on Agora objects is not strictly necessary, since we can replace it by *send*('*down*') which is supposed to invoke a 'downing method'. However, for efficiency reasons we implemented *down* directly on Agora objects.

form of an Agora object. That is, sending `QUOTE` reifies the underlying parse tree as an object in Agora (i.e. an object understanding *send*). Internally, this is accomplished by sending *up* to the expression. Hence, internally, each expression object understands the reifier message $quote(c)$, just like identifiers understand the message $variable(c, expression)$. The implementation of *quote* is to return the receiver as an upped object, i.e. "$this.up()$" yielding an Agora object that understands *send*.

The opposite of quoting is called unquoting. In Agora, this is accomplished by sending `UNQUOTE` to an expression. The receiver of this reifier must evaluate to an expression object. The resulting expression object will be downed (yielding a real expression understanding *eval*) which can then be evaluated in the context of unquoting. Hence, the implementation of the $unquote(c)$ reifier on expressions is "$this.eval(c).down().eval(c)$".

These two reifiers are particularly interesting for extending objects from the outside. Extension from the outside is accomplished by a view or mixin that does not list the new slots itself, but takes a parameter being the expression to be evaluated in the view. By unquoting this parameter, the parameter is evaluated (yielding a quoted expression), and this expression is then evaluated in the context of the view or mixin:

```
mySlots  LOCAL VARIABLE: ({ ...slots ...} QUOTE);
myObject LOCAL VARIABLE:
          { ...
            extend:slots VIEW: (slots UNQUOTE)
          };
myObject: (myObject extend:mySlots)
```

But if we can extend objects 'from the outside', then what about strong encapsulation? The answer is that objects are by default strongly encapsulated, but *can* be extended from the outside if *they* want to. In other prototype-based languages with a richer MOP, objects *cannot* avoid that they are subject to extension. This is important with encapsulation [15] and security [6] in mind.

6.2 Other Applications

Several other applications of the reflective Agora kernel exist.

First, Agora allows us to write our own reifiers. Although not very difficult, this falls beyond the scope of the paper because it requires a more technical understanding of the evaluator. See the language manual for more details [5].

Finally, it is possible to achieve structural and behavioural reflection using the `UP` and `DOWN` reifiers. Sending `UP` to an expression evaluates the expression and returns the result as a meta object (i.e. an object understanding `send:args:`). `DOWN` evaluates an expression to a meta object, and returns the object as an ordinary Agora object. The difference between `UP` and `QUOTE` is that `UP` evaluates its receiver while `QUOTE` doesn't. The meta level object returned by `QUOTE` understands `eval:` instead of `send:args:`. The difference between `DOWN` and `QUOTE` is that evaluating the receiver of `DOWN` must yield a meta object understanding `send:args:` that can be brought to the down level. Evaluating the receiver of `QUOTE` on the other hand must yield a meta object understanding `eval:` that can then be evaluated by the `UNQUOTE` reifier. In [13],

several applications of these reifiers are explained such as tracing message sends. Explaining these would lead us too far because it requires more technicalities to be understood. These can be found in the language manual [5].

7 Conclusion

In this paper, we have shown how a 'very Scheme like' OOPL can be constructed with a MOP that only knows *send*. Although not possible at first sight, such a semantic architecture still allows features like cloning and object extension by a clever usage of special purpose attributes that can access an object's internal details. Other OOPLs either introduce different language values like classes to accomplish this, or, take only objects, but then define a multitude of meta level operations on these objects.

In the second part of the paper we have shown how adding the *up* and *down* message to our MOP extends the basic language to a fully reflective prototype-based language. To some people this seems strange: if one cannot do more at the meta level than at the base level (i.e. message passing), then why adding a meta level in the first place? Agora refutes this criticism by consistently mapping every base level message onto a meta level message, and the other way around. As such, every implementation level message can be intercepted (and thus reprogrammed!) at the Agora level. This makes Agora a fully reflective language kernel in which both structural and behavioural reflection are possible.

Acknowledgements I would like to thank Prof. T. D'Hondt for promoting my work. Furthermore, I thank Patrick Steyaert and the former Agora group for starting the Agora project. Also thanks to Werner Van Belle, Tom Tourwe, Kim Mens, Kris De Volder, Daniel Bardou and the anonymous referees for commenting on earlier draft versions of this paper.

References

[1] H. Abelson, Sussman J.C, and Sussman J. *Structure and Interpretation of Computer Programs (2nd edition)*. MIT Press, 1996.

[2] G. Blaschek. *Object-Oriented Programming with Prototypes*. Springer-Verlag, 1994.

[3] C. Clinger and J. Rees. Revised(4) Report on the Algorithmic Language Scheme. In *ACM Lisp Pointers IV (July-September)*, 1991.

[4] W. Codenie, De Hondt K., Steyaert P., and D'Hondt T. Agora: Message Passing as a Foundation for Exploring OO Languages. *ACM Sigplan Notices*, 29(12):48, December 1994.

[5] W. De Meuter. Agora98 Language Manual. Technical Report, Programming Technology Lab, Vrije Universiteit Brussel (progwww.vub.ac.be), 1998.

[6] W. De Meuter, T. Mens, and P. Steyart. Agora: Reintroducing Safety in Prototype-Based Languages. In *2nd Workshop on Prototype-Based Programming (ECOOP96 - Linz - Austria)*, 1996.

[7] S. Liang, P. Hudak, and M. Jones. Monad Transformers and modular interpreters. In *Conference Record of POPL'95:22st ACM Symposium on Principles of Programming Languages*, page 472. ACM Press, 1995.

[8] C. Lucas and P. Steyaert. Modular Inheritance of Objects Through Mixin-Methods. In *Proceedings of the JMLC94 Conference*, ACM Sigplan Notices. ACM Press, 1994.

[9] J. Malenfant. On the Semantic Diversity of Delegation-based Programming Languanges. In *Proceedings of the OOPSLA'95 Conference*, ACM Sigplan Notices, page 215. ACM Press, 1995.

[10] K. Mens, K. De Volder, and T. Mens. A Layered Calculus for Encapsulated Object Modification. Technical Report, Programming Technology Lab, Vrije Universiteit Brussel, 1996.

[11] T. Mens, K. Mens, and P. Steyaert. OPUS: a Formal Approach to Object-Orientation. In *Proceedings of Formal Methods Europe (FME'94)*, pages 326–345. Springer-Verlag, 1994.

[12] T. Mens, K. Mens, and P. Steyaert. OPUS: a Calculus for Modelling Object-Oriented Concepts. In D. Patel, Y. Sun, and S. Patel, editors, *Proceedings of the 1994 International Conference on Object Oriented Information Systems*, pages 152–165. Springer-Verlag, 1995.

[13] P. Steyaert. *Open design of object-oriented languages, a foundation for specialisable reflective language frameworks.* PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, 1994.

[14] P. Steyaert, W. Codenie, T. DHondt, K. De Hondt, C. Lucas, and M. Van Limberghen. Nested Mixin-Methods in Agora. In *Proceedings of the ECOOP'93 7th European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, page 138. Springer Verlag, 1993.

[15] P. Steyaert and W. De Meuter. A Marriage of Class and Object-based Inheritance Without Unwanted Children. In *Proceedings of the ECOOP'95 9th European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, page 127. Springer Verlag, 1995.

[16] A. Taivalsaari. *A Critical View of Inheritance and Reusability in Object-oriented Programming.* PhD thesis, Department of Computer Science, University of Jyvaskyla, 1993.

[17] D. Ungar and R. Smith. Self: The Power of Simplicity. In *Proceedings of OOPSLA '87 Conference*, ACM Sigplan Notices, page 227. ACM Press, 1987.

[18] M. Wand and D. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. In P. Maes and D. Nardi, editors, *Meta-Level Architectures and Reflection.* 1988.

[19] P. Wegner. Dimensions of Object-Based Language Design. In *Proceedings of the OOPSLA'87 Conference*, ACM Sigplan Notices, page 168. ACM Press, 1987.