Higher Order Messaging

Marcel Weiher
British Broadcasting Corporation
metaobject Ltd.
marcel@metaobject.com

Stéphane Ducasse
Language and Software Evolution Group
LISTIC — Université de Savoie, France
stephane.ducasse@univ-savoie.fr

ABSTRACT

We introduce Higher Order Messaging, a higher order programming mechanism for dynamic object-oriented languages. Higher Order Messages allow user-defined message dispatch mechanism to be expressed using an optimally compact syntax that is a natural extension of plain messaging and also have a simple conceptual model. They can be implemented without extending the base language and operate through language bridges.

Categories and Subject Descriptors

D.3.3 [**Software Engineering**]: Language Constructs and Features—*Classes and objects, Control structures, Patterns*

General Terms

messages, methods, higher order messaging

1. INTRODUCTION

The initial impulse for creating Higher Order Messaging (HOM) was a desire to escape the loop ”pattern” [18], the processing of collections through repetitive use of iterative idioms in Objective-C and similar languages. The mechanism presented not only solves the original problem in the context of Objective-C it also works well in other applications and also defines a purer higher order programming mechanism for languages such as Smalltalk that already have some sort of high-order function facility.

The contributions of this article are the analysis of the current state of the art of iteration specification in object-oriented languages, the presentation of Higher Order Messaging and the illustrations in Objective-C.

The paper is structured as follow: we start by showing that loops are not well integrated in the object-oriented paradigm, then we present the concept of high-order messaging and its applications in various domains. Finally we discuss the implementation of higher order messages in Objective-C and compare our work in the context of existing languages.

Dynamic Languages Symposium (DLS) ’05, October 18, 2005, San Diego, CA, USA Copyright 2005 ACM ISBN 1-59593-283-6/05/10...$5.00

2. COLLECTION ITERATION PROBLEMS

In this section we illustrate the heart of the problem: the lack of a clean and uniform integration of control structures such as loops into object-oriented programming. Whereas object-oriented programming defines operations as messages sent to objects, control structures need additional *ad-hoc* mechanisms in most of the languages. These additional mechanisms complicate the solution to the problem at hand and add unnecessary constraints as we present now.

2.1 Objective-C

As higher order messages have first been implemented by the first author in Objective-C, we briefly recap the most important syntactical points of Objective-C. Note however that higher order messages are not specific to Objective-C and can be introduced to any dynamically-typed object-oriented languages such as Smalltalk or Ruby.

Objective-C is a hybrid object-oriented programming language based on C that adds dynamic message passing from Smalltalk to its C substrate [4]. The syntax is a strict superset of C that adds a message send operation, delimited by square brackets and using Smalltalk keyword syntax *i.e.,* arguments are placed inside the method name preceeded by a colon “:”.

```
[employee salary];
[employee setSalary:10000];
[employee setFirstName:@”Sally”];
[[employee manager] salary];
```

String object literals are prefixed with the @ character, for example @”I am a string” is the string ’I am a string’. There is additional syntax for class-definition and declaration that we will not use here. The @selector() construct is used to specify the name of a message that can be used to send programatically a message. For example, the following two expressions both send the message addObject: to the salariedEmployees object, the first directly, the second using reflective facilities:

```
[salariedEmployees addObject:anEmployee];
[salariedEmployees
        performSelector:@selector(addObject:)
        withObject:anEmploye];
```

Class definitions come in two parts, an interface, which declares the instance variables very much like a C struct would and publicly visible methods, and an implementation section. Each section is terminated with the token @end. Following C conventions, interfaces are usually placed in header files, but this is not a requirement, a file can contain as many interface and implementation sections as desired.

```
@interface MyClass : NSObject {
    int counter;
}
-(void)addOneToCounter;
@end
```

Class interface in Objective-C

The classed declared in the interface shown above has on instance variable of type integer and declares one method with no return value, which is implemented in the following, corresponding implementation section:

```
@implementation MyClass
-(void)addOneToCounter
{
   counter++;
}
@end
```

Method implementation for class in Objective-C

In addition to NSObject, the standard root class for the Cocoa frameworks, this paper will also reference NSProxy, another root class that implements only the bare minimum methods required for interacting with the Objective-C runtime and NSInvocation, an object representing a message send with all its components: the receiver, the selector and the arguments.

2.2 A Collection Processing Example

Let's say we have a collection, employees, containing employee objects and we want to find out which of these employee objects earn more than 1000 Euros. A solution is shown by the following piece of code. First we create a collection that will contain the employees meeting our requirement, then we iterate over the collection and for each of the element we check the condition and add the current element to the resulting collection.

```
int  i;
id  salariedEmployees = [NSMutableArray array];
for (i = 0 ; i < [employees count] ; i++ ){
    id employee = [employees objectAtIndex: i ];
     if ( [employee hasSalary: 1000]  ) {
         [salariedEmployees addObject: employee];
    }
}
```

The indexed selection loop.

This code is virtually the same *every* time we want to select objects from a collection based on the return value of a message. In fact, the only parts that vary are the parts highlighted in bold, the rest is boilerplate code for the pattern "select elements from a collection", with the parts that generically implement selection in italic and the parts that iterate through the collection as plain text. This code is not only redundant and error-prone [20], it also acts as "...an intellectual bottleneck that [keeps] us tied to word-at-atime thinking instead of encouraging us to think in terms of larger conceptual units of the task at hand" [1].

Possible Solutions. Using the Iterator Pattern [8] makes the code a bit less error prone and insulates it from the actual type of collection used, but doesn't remove much redundancy, still introduces additional concepts and elements that are not directly related to the original task and forces the developer to deal with single instances individually. In fact, actual code bulk is increased compared to using array indexing because language syntax typically does not directly support iterators as shown by the following code.

```
id  nextEmployee;
id employeeIterator = [ employees objectEnumerator];
id  salariedEmployees = [ NSMutableArray array ];
while ((nil != (nextEmployee = [ employeeIterator nextObject ])){
     if ( [ nextEmployee   hasSalary: 1000 ]  ) {
             [ salariedEmployees addObject:next];
    }
}
```

The enumerator selection loop.

Adding special syntactic support for iterators slightly reduces the code bulk for looping but does not improve on the boilerplate code for selection, and still leaves the client to iterate sequentially over the individual elements of the collection and deal with single instances as shown by the following Java 1.5 code.

```
NSMutableArray  salariedEmployees=new NSMutableArray();
for ( Employee each :  employees ){
      if (  each. hasSalary( 1000 )   ) {
            salariedEmployees.addObject( each );
      }
}
```

Java for-each loop

Smalltalk [10] goes a lot further in allowing us to eliminate boilerplate code, having ready-made and reusable iteration constructs. However, following the lisp tradition of map, we have to wrap the message we want to send in a Smalltalk block *i.e.*, a lexical closure, and we also still have to mention individual elements, despite the fact that we are interested in the collection as a whole. The same problem is expressed as follow in Smalltalk.

```
salariedEmployees:=employees select:[:each | each hasSalary:1000]
```

Smalltalk select with a block

Problems summary. All of the above solutions suffer in varying degrees from not being able to directly express the intent of the operation, which is to create a subset of the original collection based on some criteria. Instead, we have to deal with irrelevant details of accessing and processing individual elements that obscure the intent of the operation, add possibilities for error and impose unnecessary constraints on the solution.

Higher order messages in a nutshell. Using Higher Order Messaging removes all the boilerplate code: the result is obtained by

$$curried(x)(y)(z)$$

$$normal(x, y, z)$$

Figure 1: Curried vs. normal function

selecting all the elements from array for which the message hasSalary: 1000 return true. In addition, the array is, at least conceptually, treated as a whole unit, client code never has to reference or name individual elements following the APL view of arrays [18]. However HOM is not limited to collection operation and can be applied to a large family of problems as we show later.

```
salaried=[[employees select] hasSalary: 1000] ;
```

Select with Higher-Order Messages

Furthermore,since no additional elements are required, there are no additional constraints imposed on the implementation of either the collection or the iteration process.

3. HIGHER ORDER MESSAGES

A *Higher Order Message* (HOM) is a message that takes another *message* as its argument, allowing the Higher Order Message to control the delivery of the argument message. However, Objective-C and other object-oriented languages don't support sending a message as a parameter to another message, not even syntactically [6]. What is supported, however, is messages taking objects as arguments, so we need to somehow bridge this gap.

Currying. In order to bridge this gap, we borrow a technique from functional programming called *currying*. Currying takes a function with n parameter and turns it into a function of 1 parameter that returns a function of $n-1$ parmeters. In other words, instead of passing parameters into a function, additional arguments are processed by sequentially applying the results of previous function invocations to new arguments (see figure 1).

Syntax. Syntactically a Higher Order Message expression takes the form of a *prefix message* (semantically: the higher order message), followed by the curried *argument message*, as shown in figure 2

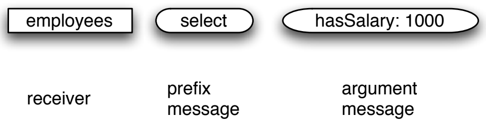


Figure 2: Curried argument message

Higher Order Messaging borrows the concept of currying and adapts it to message sending in object oriented programming languages: instead of passing the argument message to the prefix message directly, which isn't possible, the prefix message returns an object that is then sent the argument message. This object then reifies the argument message it was sent and goes back to send a *worker* message to the original receiver, as shown in figure 3.

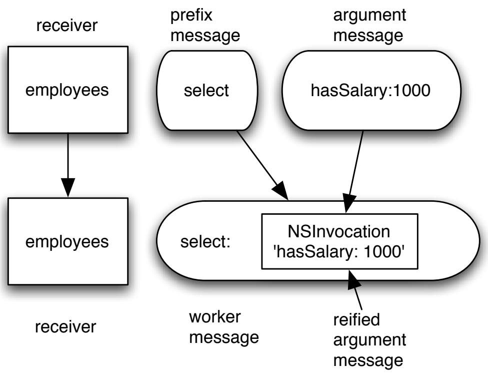


Figure 3: Uncurrying

This allows the existing message sending syntax and mechanisms provided by the base language to be used directly to specify the argument message, instead of having to construct message objects manually. This comes at the expense of having to implement both a *prefix* method handling the curried to uncurried conversion as well as a *worker* method handling the actual processing using the reified argument message.

Semantics. In this example, the prefix message select is the higher order message and the curried argument message is hasSalary:. The higher order message specifies routing for the argument message and its return values, if any, so it is the higher order message that determines the semantics of the return of the complete expression. Here the collection of the selected elements is returned.

The type of routing performed depends on the higher order message used, and as higher order messages are user-defined using the full power of the underlying programming language, there are effectively no limits. The select HOM used in this particular example returns a new collection containing all the objects from the original collection that return true to the argument message, in this case the objects that have a salary higher than 1000 Euros.

Higher Order Messages takes care of converting between the convenient curried form that has the argument message written using plain message syntax and the non-curried form that is required for actual processing of the Higher Order Message.

HOM is not limited to selecting collection elements. In the following section, we present a couple of uses that HOM has been put to, such as more general collection processing, lazy computation, asynchronous, cross-thread and delayed messaging and various other utilities.

4. HOM APPLICATIONS

An implementation of Higher Order Messaging itself is only a tiny addition to an object-oriented language, and typically fully implementable in library code without requiring changes to the language. However, because of the natural way it interacts with and extends messaging, the applications and implications of the concept are quite far-reaching, as we hope to show with a few sample applications of the technique.

4.1 Collections

Collections are an area where the limitations of first order messaging become readily apparent. On the one hand, a collection is an object that has attributes and behavior of its own, but on the other hand a collection is also a reference to the objects it contains, similar to the way a variable is a reference to the object it contains, and just like we typically want to refer to the value of a variable, we often want to refer just to the contents of a collection.

Higher Order Messaging provides a way for differentiating between messages sent to the collection object itself and to the contents of the collection. The Higher Order Message specifies that the argument message gets routed to all the individual objects that make up the contents of the collection, and also what, if any, processing is to be applied to the return values.

Collect. The collect HOM provides the functionality of the map higher order function that returns all the function application results. Given a list of names, say @"John", @"Mary" and @"Bob", the following piece of code appends @" Doe" to each of the names, yielding @"John Doe", @"Mary Doe" and @"Bob Doe":

```
names = [NSArray arrayWithObjects:@"John",@"Mary",@"Bob",nil];

[[names collect] concat: @" Doe"][1]
```

Varying receiver

Natural order. However, we are not limited to iterating over the receiver of the argument message. Let's take the example of an employee who is a manager and therefore has reports, people that report to her, assigned to her. To add a new report, there is an addReport: method, so if alice is a manager and sally is to be added to her direct reports, that is to the list of people that report to her, you write the following:

```
alice addReport:sally.
```

Adding a single report

The structure of the code mimics natural english sentence structure: "Alice adds Sally to her reports". In natural language, this sentence structure remains the same not matter who we add to Alice's reports. For example, if Alice adds Bob to her reports, the sentence would be: "Alice adds Bob to her reports".

However, imagine Sally leaves the company, and therefore we want to add all of her reports to Alice's reports. We would write this as: "Alice adds Sally's reports to her reports". In general the sentence structure is "Alice adds *someone* to her reports" and remains that way no matter who *someone* actually is.

However, in most programming languages, unlike in natural languages, the fact that we are now adding a collection of people to Alice's reports changes the entire structure of the expression, because we are forced to deal with the iteration aspect first and foremost. So in Smalltalk, for example, adding Sally's reports to Alice's reports is implemented as follows:

[1]The actual message used in Cocoa is **stringByAppendingString:**, it is abbreviated to **concat:** here in order to ease formatting.

```
sally reports do: [ :each | alice addReport:each ].
```

Adding many reports with iteration

The subject of the whole expression, alice has now moved somewhere into the middle, surrounded by all sorts of machinery that makes the iteration go. With HOM, the subject and the verb (the message) remain in place, in fact the whole expression is unchanged except for the part that actually needs changing, the object (grammatically speaking) of the expression. The fact that there may be iteration going on is put in the background, because it really isn't important, all we are interested in is that something happens to that group of objects.

```
alice do addReport: sally reports each.
```

Adding many reports with HOM

Figure 4 shows how the structure of an expression completely changes when moving from a single argument to a collection of arguments without HOM. All the elements shift around, reversing the original order and focus is now placed on what is actually the argument of the expression, and the expression itself is moved inside the block required for the iteration.

In contrast, Higher Order Messaging retains all the original elements in the original order.

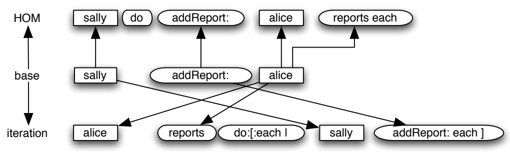


Figure 4: Going from single to collection of arguments

Iterating over arguments. The previous example used the each message to request iteration over an argument of a message. This ability to iterate over any combination of arguments is a feature of the specific implementaton of the collect and select iteration Higher Order Messages in MPWFoundation, an open source Objective-C framework that implements HOM as presented in this paper.

The each message wraps its receiver in a special marker object that signals to the iteration implementation that we want to iterate over this collection, that we are interested in the contents of the collection rather than the collection object itself. In the current implementation, that marker is just an iterator over the collection.

The loop implementing the iteration iterates through all iterators it finds in the argument message simultaneously, which is possible because the iterators happen to be external iterators. It performs this simultaneous iteration by constructing a new message, the *loop message* with a new argument list from the original message during each pass through the loop.

For *constant arguments*, the ones that do not change during the iteration, the argument is simply copied from the original message to the loop message. If all arguments are constant this results in a single message being sent with exactly the same arguments as the original messages.

Variable arguments, however, meaning the ones that are to be iterated over and therefore marked by the iterators created with the each message, are treated differently. When the loop encounters an iterator in the original argument list, it does not copy the iterator itself to the loop message. Instead it gets the next object from the iterator and copies the object obtained to the loop message.

In this way, one new object is obtained from each iterator present in the original message that was the argument of the Higher Order Message. The loop terminates as soon as any one of the iterators is exhausted.

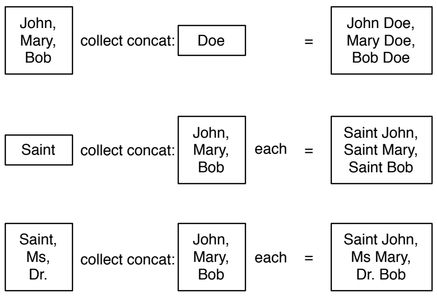


Figure 5: collect with varying receiver, argument or both

By allowing iteration over any combination of receiver and arguments, this simple algorithm also makes it both easy and natural to iterate over multiple arguments of the same length simultaneously, something that is difficult or simply impossible with other iteration constructs.

The reason this is possible is that all parameters to the argument message are passed using the same parameter passing mechanism. The following example (results shown in Figure 5) concatenates each prefix with the corresponding suffix:

```
[[titles collect] concat: [names each]]
```

Mulitple argument iteration

Attribute queries. The introduction already showed the use of the select higher order message to select objects from a collection based on their response to a predicate message such as hasSalary:. Sometimes, however, we also need to select based not on a predicate on the object itself, but based on a predicate of an attribute[2] of the object. For example, we might want to select all employees whose name is John, similar to the following SQL query:

```
select * from employees where name = ”John”
```

SQL select employees based on name

[2]Of course, we do not deal with attributes directly, but with any value that can be obtained by sending a message to the object in question

While it is theoretically possible to add a hasName: predicate message to use directly with the select HOM, this quickly becomes infeasible. For example, if we wanted to select employees based on wether their names started with John, we would have to add yet another message, nameStartsWith:.

Instead, we add the selectWhere HOM, which differs from the HOMs introduced thus far by taking *two* argument messages. The first argument message is the message that retrieves the attribute to be queried, the second argument message is the predicate to use on that value. Using selectWhere, the name-based queries can be expressed as follows:

```
[[[employees selectWhere] name] isEqual: @”John”]
[[[employees selectWhere] name] startsWith:@”J”]
```

query on attributes with selectWhere

There are several ways of achieving more complex queries. The simplest is to add a method encapsulating that complex query to the class in question, which has the benefit of providing a name for that query. Additionally multiple HOMs can be nested in order to achieve the desired effect. If that isn't sufficient, it is possible to implement a HOM that does not trigger processing immediately, but rather accumulates messages into a complex query object. This then requires a trigger message to indicate that accumulation of the query is to end and processing should commence.

4.2 Multiprocessing

The higher order iteration messages introduced above express the intent of processing all the elements of a collection without the programmer having to pollute that specification with any sequential ordering information such as an explicit loop. It is therefore easy to execute these operations in parallel without the need for a smart compiler to unpick the arbitrary order constraints imposed by normal iteration code.

Message scheduling. The OPENSTEP standard class library for Objective-C, popularized by Cocoa, has facilities for scheduling messages on threads or for later execution. The following samples show the message-send [object doSomething] performed in a separate thread or scheduled after a delay of 1 second:

```
[NSThread detachNewThreadSelector: @selector(doSomething)
        toTarget: receiver
        withObject:nil ]
```

Dispatch a message to an object in a new thread

This expression schedules the message message doSomething to be sent to the receiver in the context of a new thread. Although there are no arguments to send with the doSomething message, we have to provide a dummy 'nil' parameter because only one version of the detachNewThreadSelector: toTarget:withObject: is provided by the framework. If we had wanted to send a message with more than one argument, or a non-object argument, we'd be out of luck.

```
[receiver performSelector: @selector(doSomething)
            withObject:nil
            afterDelay:1.0 ]
```

Dispatch a message to an object after a delay of 1 second

This example sends the message doSomething to receiver after a delay of 1.0 seconds. As in the previous example, we have to provide a dummy 'nil' argument and would not be able to send a message with more than one argument.

Both examples suffer from the fact that the argument message send [receiver doSomething] is completely obscured by being split up as arguments and intertwined with message specifying special treatment.

With higher order messages, the argument message and the routing message are clearly separated, and the expression much more clear and succinct:

```
[[object async] doSomething]
[[object afterDelay:1.0] doSomething]
```

Dispatch in thread or after delay with HOM

The original message send is clearly visible following the prefix message, as is the extra scheduling specified in the prefix message.

In Smalltalk detaching a thread is typically done with a block:

```
Process fork: [receiver doSomething]
Process fork: [(Delay forSeconds:1.0) wait. receiver doSomething]
```

Dispatch in thread or after delay with Smalltalk blocks

4.3 Futures

Futures are a hybrid solution between a synchronous and asynchronous message send. They appear to return a result immediately, but in fact that result is being computed asynchronously [25, 26]. Synchronization only occurs when the result is actually used.

Multilisp introduced the future mechanism [24] and also a simple syntax for indicating future-processing that makes extending a specific expression to parallel processing straightforward. The following code shows the merge phase and recursive call of a mergesort both in serial and parallel versions:

```
(merge (mergesort x) (mergesort y))
(merge (future (mergesort x)) (mergesort y))
```

mergesort and future-mergesort in Multilisp

The only difference for the programmer is to wrap a call to future to the function call that should run in parallel. The reason it is as simple as that is that the future() function can take entire function calls as an argument and that function calls are the natural way to express a computation in LISP.

Although other published implementations of futures [23] strive for similar ease of expression, they fall short despite including a preprocessing step. HOM naturally achieves the same ease of expression as Multilisp, for pretty much the same reason.

```
[[x mergesort] merge: [y mergesort]]
[[[x future] mergesort] merge: [y mergesort]]
```

mergesort and future-mergesort with HOM

With Higher Order Messaging, futures can be used as if they were fully integrated into a language's messaging system, and the same code also gives us lazy evaluation, as will be shown in the discussion of the future implementation in section 5.2

4.4 Utilities

Higher Order Messages are quick to implement, and therefore lend themselves to making various repeating patterns of code more compact, expressive and readable, as demonstrated by the various smaller examples shown below.

Delegates. Cocoa makes frequent use of the delegate pattern to allow unrelated objects to be integrated into the framework's processing. In order to make this happen, methods need to check whether a delegate actually responds to the message it is about to be sent. The following piece of code presents a common Objective-C idiom: first it checks whether the object can understand the message windowDidClose: and if this is the case send it.

```
if ( [delegate respondsToSelector:@selector(windowDidClose:)] {
   [delegate windowDidClose:self];
}
```

Check delegate can respond before sending

The duplication of the message selector, first in the check and later in the send not only means more effort in writing the code and greater difficulty reading and understanding it, it is also a potential source of errors if the two instances don't match, for example due to a typo or a change that was only carried out, such as in the following code that checks for windowDidClose; but then goes on to send windowWillClose:.

```
if ( [delegate respondsToSelector:@selector(windowDidClose:)] {
   [delegate windowWillClose:self];
}
```

Buggy delegate check for wrong message

Using an ifResponds HOM, this is reduced to the following:

```
[[delegate ifResponds] windowDidClose: self];
```

Send message if it will be understood

The message only occurs once, therefore eliminating the possibility of checking the wrong message.

Exception handling. Exception handling code also has distinct patterns, for example it is often desirable ignore an exception at a particular point in order not to cut short further processing, yet still log the exception in order to leave a record of the exceptional event:

```
@try {
   [receiver sendMessageThatMightThrow];
} @catch( NSException *e) {
     NSLog(@"exception: "%@",e);
}
```

Ignore and log exception

Instead of repeating this code wherever this is desirable, capturing it in a HOM allows us to write the following:

```
[[receiver logAndIgnoreException] sendMessageThatMightThow];
```

Ignore and log exception with HOM

There are many further examples of this sort, such as variations of exception logging, logging message sends, converting exceptions to return codes and vice versa, and variations of assured resource reclamation similar to C#'s using statement. The idea is always the same: factor recurring patterns of code into a named Higher Order Message in order to make code more compact and easier to understand.

5. IMPLEMENTATION

When discussing the HOM implementation, we have to distinguish between specific Higher Order Messages and the HOM mechanism itself as supplied by the framework. We present this mechanism first and then show how it can be used to create specific Higher Order Messages.

The main task of the HOM framework code is to convert/reify argument messages from their curried form as instructions to send a message to actual message objects that can be processed by user provided code.

The message-reification process relies on a feature of the Objective-C runtime system that is very similar to Smalltalk's #doesNotUnderstand: processing [7]: when the runtime does not find a method for a particular message, it reifies the message and sends the receiver a new message forwardInvocation: with the reified message as the single argument to whom? the receiver to the original message

HOM uses this mechanism by returning a temporary object called a trampoline from the prefix message, an object which implements (virtually) no methods of its own, and will therefore have no implementation for the argument message. When the argument message gets sent to the trampoline, the argument message is therefore not understood by the trampoline, meaning that the runtime cannot find a method in the trampoline's class implementing that message.

The runtime reacts to this message lookup failure by first reifying the argument message it tried and failed to deliver, and then sending the trampoline the forwardInvocation: message with this reified argument message. The trampoline takes that message object, which represents the argument message, and passes it to a target object using a *worker message*. The trampoline is set up beforehand with the target (often theoriginal receiver) and the worker message, and when it receives the forwardInvocation: message it simply forwards ('bounces') the reified message back to the target using the worker message.

MPWTrampoline is a helper class provided by the framework that encapsulates message capture using the -forwardInvocation: runtime hook, shown below. It reproduces the well-known message passing reification Smalltalk happening on not understood messages (doesNotUnderstand:).

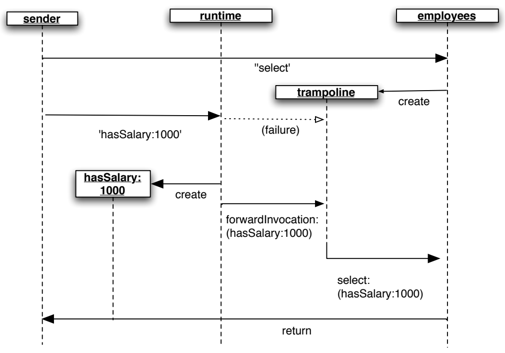


Figure 6: Sequencing for [[employees select] hasSalary:1000]

```
@interface MPWTramponline :NSProxy {
    SEL  xxxSelector;
    id      xxxTarget;
}
-initWithTarget:aTarget message:(SEL)aSelector;
@end

@implementation MPWTramponline
-initWithTarget:aTarget message:(SEL)aSelector
{
   if  ( nil != (self=[super init]) )  {
      xxxTarget = aTarget;
      xxxSelector = aSelector;
   }
   return self
}

-(void)forwardInvocation:(NSInvocation*) invocationToForward
{
   [xxxTarget performSelector:xxxSelector
                  withObject:invocationToForward];
   [self setXxxTarget:nil];
}
```

Implementation of MPWTrampoline forwarder class

As we want to use curried argument messages, yet curried messages are not directly supported in the language, an actual Higher Order Message usually takes two methods to implement: the prefix message to convert the curried argument message and the worker method to process the uncurried argument.

Due to the fact that HOM is based purely on messaging, it also works well through language bridges.

5.1 Implementation of the ifResponds HOM

Shown below is the implementation of the ifResponds Higher Order Message, which sends to the receiver its argument message if the receiver responds to that particular message, and does nothing if the receiver does not respond to the message.

```
-ifResponds
{
    return [[[MPWTrampoline alloc] initWithTarget:self
                                      message:@selector(ifResponds:)];
}
```

```
-ifResponds:(NSInvocation*)argumentMessage
{
    id result=nil;
    if ( [self respondsToSelector:[argumentMessage selector]] ) {
        [argumentMessage invokeWithTarget:self];
        [argumentMessage getReturnValue:&result];
    }
    return result;
}
```

Implementation of prefix and worker methods for ifResponds HOM

5.2 Sample HOM implementation of futures

A future computation is implemented by starting the computation to be performed in a separate thread, while at the same time returning a proxy object that will stand in for the result while it is being computed. This proxy will forward all messages to the result once the result is computed. If the result is not available, the proxy will block until it is.

```
@interface MPWFuture : NSProxy {
   id                target;
   NSInvocation*       invocation;
   id                _result;
   NSConditionLock*    lock;
   BOOL              running;
}

+futureWithTarget:newTarget;
-(void)runWithInvocation:(NSInvocation*)invocation;
-result;

@end

@interface NSObject(future)
-future;
@end

@implementation MPWFuture

+futureWithTarget:newTarget
{
   return [[[self alloc] initWithTarget:newTarget] autorelease];
}

-initWithTarget:newTarget
{
  [self setLock:[[[NSConditionLock alloc] initWithCondition:0] autorelease]];
   [self setTarget:newTarget];
   return self;
}

-(void)invokeInvocationOnceInNewThread
{
   id pool=[NSAutoreleasePool new];
  [self setResult:[invocation returnValueAfterInvokingWithTarget:target]];
   [lock unlockWithCondition:1];
   [pool release];
}

-(void)startRunning
{
   running=YES;
   [NSThread detachNewThreadSelector:
                  @selector(invokeInvocationOnceInNewThread)
                   toTarget:self withObject:nil];
}

-(void)lazyEval:(NSInvocation*)newInvocation
{
   NSInvocation *threadedInvocation=[newInvocation copy];
   [self setInvocation:threadedInvocation];
   [threadedInvocation release];
   [newInvocation setReturnValue:&self];
}

-(void)futureEval:(NSInvocation*)newInvocation
{
   [self lazyEval:newInvocation];
   [self startRunning];
}

-(void)waitForResult
{
   if (!running) {
      [self startRunning];
   }
   [lock lockWhenCondition:1];
   [lock unlock];
}

-result
{
   if ( ![self _result] ) {
      [self waitForResult];
   }
   return [self _result];
}

-(void)forwardInvocation:(NSInvocation*)messageForResult
{
   [messageForResult invokeWithTarget:[self result]];
}

@end
```

The MPWFuture class

Note that MPWFuture also makes use of forwardInocation: to handle messages sent to it, similar to MPWTrampoline, so it is important to keep these two uses of the mechanism separate. Also note that future evaluation is actually based on lazy evaluation. The future mechanism starts by initializing a lazy-evaluation, which would start evaluation once the value is requested, but then actually triggers the lazy evaluation in a separate thread.

The actual Higher Order Message is implemented using the pattern for prefix messages:

```
@implementation NSObject(future)

-future
{
   id future = [MPWFuture futureWithTarget:self];
   return [MPWTrampoline trampolineWithTarget:future
              selector:@selector(runWithInvocationInNewThread:)];
}

@end
```

The future Higher Order Message

5.3 Performance

The current implementation of Higher Order Messaging imposes a small up-front performance overhead for converting the message from its curried form to an actual argument message. This overhead is primarily composed of the message lookup failure and the cost of allocating and initializing both a trampoline and a message object from the argument message send, with the object allocation overhead being the primary factor in Objective-C, which relies on comparatively slow malloc() based object allocation.

After the initial overhead, performance depends very much on the implementation of the particular Higher Order Message. The collection processing HOMs implemented in MPWFoundation use techniques for optimizing message sends in Objective-C in order to make the asymptotic performance even slightly better than a normal iterator-based loop.

Array size	loop (μs)	collect HOM (μs)	relative (%)
10	3.70	8.10	218
100	1.53	1.87	122
1000	1.27	1.25	98
10000	1.31	1.23	94

Table 1: Processing time per element

Table 1 shows results that were gathered running on a PowerBook with an 867MHz G4 running at full speed. A 'do-nothing' message was used to generate results from an array using either a collect Higher Order Message or an equivalent iterator-based loop. This was repeated 1000 times and the times measured straight in the program with getrusage().

The results clearly show the initial overhead of about 30-40 μs that gets dwarfed by the savings from the optimized looping implementation as the array gets larger. While the optimizations that are performed inside the collection processing HOM implementation could conceivably be used when writing plain loops, the extra coding effort and decrease in code clarity usually do not make it worth the effort. However, the fact that a single loop implementation will be reused many times over makes that extra effort worthwhile.

Opportunities for future optimizations include eliminating the message failure overhead by turning trampolines into *learning proxies* that dynamically generate an implementation for messages that are not understood.

This technique eliminates the initial lookup failure, allows specialized code to be employed that knows exactly what parameters to gather from the stack, instead of generic code that has to examine parameters at run time using stack-crawling techniques, and finally also allows specialized replacements for NSInvocation to be used.

5.4 Real World Use

Higher Order Messaging has been implemented in MPWFoundation, an open-source Objective-C framework available for download free of charge at http://www.metaobject.com/.

It has been used in other frameworks, various open source projects, and in several commercial products that have shipped several thousand copies in the last 4 years.

6. DISCUSSION

So far, we have demonstrated how Higher Order Messaging elegantly solves various problems of flow control for object oriented programming languages. Iteration has been a particular problem, with many languages simply borrowing structured control flow mechanism from ALGOL-like structured programming languages.

These mechanisms are foreign imports into the object oriented paradigm, which posits that computation occurs via messages sent to objects. It is therefore not entirely surprising that these mechanism cause exactly the sorts of problems that OO was supposed to address: weak encapsulation, inline expansion of code, lack of reuse, non intention-revealing code, no user-extensibility.

6.1 Higher Order Functions

Functional Programming Languages manage to solve some of the problems of structured control flow mechanisms through the use of Higher Order Functions. Higher Order Functions allow control flow to be encapsulated, reused and parameterized with other functions that describe the work to be controlled.

Functions in functional languages have a a close analog in object oriented programming languages: methods. Therefore, applying the techniques used with Higher Order Functions to OO seems like an obvious solution to the problem of control. However, it is not actually possible to directly pass methods to objects [6,12], because a method is actually incomplete without its receiver, and it cannot be parameterized with arbitrary receivers because the receiver (at minimum) selects the method to be used.

6.2 Blocks

Smalltalk gets around this problem by introducing anonymous functions from FP in the form of blocks. This addition means that Smalltalk, although it usually considered the "purest" object oriented language, is probably more accurately characterized as a multi-paradigm language mixing elements from FP and OO.

Blocks bring many of the benefits of Higher Order Functions to Smalltalk, but also have some significant drawbacks. One is that only anonymous functions are available, whereas FPLs can use either named or anonymous functions. While it may sometimes be convenient to specify the argument function inline, it is principally undesirable as it discourages modularity and encapsulation and also prevents the use of an intention-revealing name. The workaround for this is to use a message anyhow, and then wrap that message in a trivial block, making the block an element of "accidental complexity" [3].

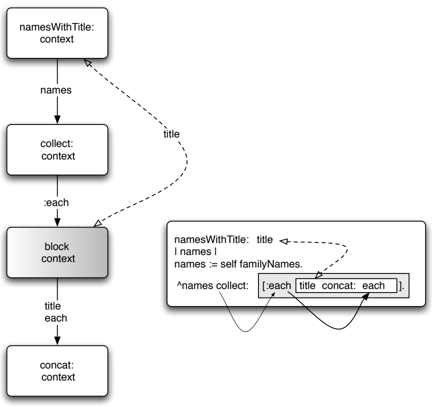
The requirement for inline specification also makes it impossible to hide the processing of the individual elements of a collection, meaning the programmer is forced to deal with collection elements one "item at a time". In addition, an extra execution context is introduced, which not only needs to be activated and torn down, but also needs parameters passed into it. In order to do this, an additional method of parameter passing, lexical variable catching via closures is required, as illustrated in Figure 7.

6.3 Messaging

Although Smalltalk has shown that it is possible to get around the problem of methods as arguments by introducing anonymous functions, another possibility is to realize that methods aren't actually the appropriate element in the first place.

Object oriented programs do not call methods, they send messages to objects. With the first order mechanism being messaging, it seems both straightforward and consistent for the higher order mechanism to also be based on messaging.

Messaging is a highly flexible term and it is a shame that even in



```
namesWithTitle:  title
| names |
names := self familyNames.
^names collect:  [:each  title  concat:  each ].
```

Figure 7: Extra execution context and non-paramater passing

dynamic OOPLs, it hardly ever means more than “invoke a selected procedure”. Messages can be sent synchronously or asynchronously, multiplexed, transmitted, repeated, broadcast, stored, forwarded, saved lost or ignored, and this list probably only scratches the surface.

Higher Order Messaging takes advantage of this flexibility of messaging by allowing one message to control the delivery of another message, therefore effecting the control flow of a program. Effectively, HOM allows the programmer to easily define libraries of message delivery mechanisms, and to conveniently choose from those message delivery mechanisms on the fly for different message sends.

6.4 Consequences

As we have seen in the preceding sections, consistently using messaging not just as our first order mechanism, but also as our higher order mechanism brings not just a theoretical benefit of conceptual simplicity and integrity (see table 2), but also practical benefits. Expressions are both compact and easy to read, they fully reveal the intension of the expression without any extraneous elements or syntactic glue.

	first order mechanism	higher order argument
FP	functions	functions
logic	predicates	predicates
OO + Blocks	messages	anonymous functions?
OO + HOM	messages	messages

Table 2: First- and higher-order concepts

Being able to combine existing code without the need for inline glue encourages modularity by bringing full benefits to such modular code, whereas a requirement for inline glue encourages the placement of actual functionality into that inline glue code. On the other hand, the fact that HOM requires a message-send can also be limiting because sometimes there is no ready-made message that captures what is needed, and the functionality doesn’t really seem significant enough to create a separate method.

The fact that Higher Order Messages are user-defined means that we can capture, encapsulate and reuse patterns of message sending as they occur in real code, rather than having to play the intellectually stimulating but unnecessary procrustean game of adapting to a fixed set of pre-existing control structures.

The fact that messaging has many different variations in the real world (think of a message being scribbled on a piece of paper, duplicated in a printing press, broadcast on a radio station, placed in a bottle, left on an answering machine, ignored etc.) makes it easy to explain fairly complex control flows even to novices.

7. RELATED WORK

There have been many attempts to bring higher order processing to object oriented languages over the years, especially for the common task of collection processing, as well as methods for redirecting messages sent to an object.

Patterns Both the Iterator and Visitor Patterns use objects to separate traversal from computation [8]. However, they require operations to be encoded as objects and classes, and therefore tend to impose significant overhead both at site of definition and use.

Foreach Both version 1.5 of the Java language and version 2.0 of the C# language add a `foreach` construct that is an extension of the basic `for` loop designed specifically to iterate over collections with an iterator. As shown in the introduction, this special language construct hardly makes a difference in the actual code bulk required for iterating through a collection, though it does make the setup a little more convenient.

Generators Generators, introduced in CLU [14] and also used in Sather [19] and Python [15] provide a very flexible and generic mechanism for iteration based on a coroutine concept that can yield values to a “calling” loop yet continue executing quasi-concurrently to yield more values. While very flexible and powerful, they still keep the focus on a word at a time.

Map Higher Order Functions such as Map originated in LISP and variations are now available not only in most if not all functional programming languages and many object oriented languages such as Smalltalk, Python and Ruby [9, 15, 16], usually in conjunction with some sort of lambda/closure mechanism.

In addition various attempts have been made to add functional programming techniques to other object oriented languages such as C++, Java, Eiffel and even Objective-C [5,12,17,21]. Without compiler modifications, these attempts require methods to be wrapped in classes, making the whole procedure somewhat cumbersome and defeating any gains in expressiveness that may have been obtained from HOFs.

FP and APL Both Backus’s *FP* calculus and Iverson’s *A Programming Language* allows direct composition of functions by functors without any reference to values whatsoever , yielding new functions that can then be applied to values [1, 11]. This processing of collections to be specified without tediously picking each individual element out of the collection, processing it and then constructing a new collection with the results.

Both APL and FP were major sources of inspiration for Higher Order Messaging, or conversely sources of dissatisfaction with the current state in collection processing.

OOPAL Object Oriented Array Programming [18] is a system that combines array programming techniques from APL with object oriented programming. It allows programmers to express operations on entire collections of objects without having to name the individual objects.

However, OOPAL currently applies only to arrays, with iteration being either implicit or specified by a special message pattern syntax. Implicit iteration makes for syntactically very compact and elegant expressions, but can also make code difficult to read, as it is not clear from the source wether a specific message will be sent to the array or to the array's contents. Message patterns allow compact specification of traversals, but lack the generality of prefix messages.

HOM is not limited to arrays because user-supplied HOMs can iterate over any conceivable data structure. In fact, HOM is not even limited to collections, but can be used to encapsulate any sort of control-flow problem expressible as a custom message sending mechanism. Furthermore, the HOM is always named and explicit at the point of call, leaving a syntactic clue that something is happening, and allowing the full use of good naming to describe the intent of the operation.

In many ways, OOPAL can be regarded as a special case, or a specific instance, of Higher Order Messaging, although HOM itself would only be the glue to connect an array processing engine such as that provided by OOPAL into the base language.

Adaptive Programming Adaptive programming attempts to isolate programs from specifics of class structures by allowing adaptive programs to navigate to specific parts of an object graph based on traversal patterns [13]. The traversal patterns are based on constraints placed on the class hierarchy. A traversal expression to compute the combined salaries of all the employees of a company could look as follows: "start at the Company representing, traverse down until you find Salary objects and then add their values".

Similar to HOM based collection processing and the visitor pattern, AP separates traversal from processing. For traversal that closely match the traversal constraints of the traversal patterns, it provides a very concise means of specifying traversals that is also robust in the context of changes to the intermediate shape of the the object and class graphs. However, traversal patterns are not suitable for general traversal/iteration specification and not suitable at all for message patterns not directly involving traversals.

Therefore, adaptive programming can probably be characterized as a specialized form of Higher Order Messaging.

Composition Filters Composition filters enhance object oriented programming by allowing filters to adjust messages arriving at or emanating from an object [2]. As such they also act on the routing of messages to objects. However, composition filters are attached to objects, not triggered by message sends. Therefore, they have to be specified statically and therefore cannot be freely composed with argument messages on the fly to achieve such as effects as collection iteration.

Natural Programming Languages The HANDS system [22] allows users to operate on aggregates as a unit, without requiring iteration over the elements. Such aggregate operations were used much more than looping, and children using the full HANDS system performed better than other children using a modified version of the system without the aggregate operations.

This research provides empirical support for the assertion made by us and others [1] that conventional iteration constructs that require users to deal with individual objects present a difficulty in programming and operating on the collection as a whole is somehow easier. The system has syntax that tries to mimik natural languages, as shown below:

```
set the nectar of all flowers to 0
```

This expression can be translated in a straightforward way using HOM (in either Objective-C or Smalltalk syntax):

```
[[flowers do] setNectar:0]
flowers do setNectar:0.
```

Other HOM implementations Initial releases of MPWFoundation and Higher Order Messaging with it have sparked a flurry of implementations with variations on the theme of Higher Order Messaging, mostly in Objective-C but also in Smalltalk.

8. CONCLUSION AND FUTURE WORK

We have introduced a mechanism for controlling program flow by dynamically controlling message sending using prefix messages that take curried argument messages. This mechanism greatly simplifies not just collection processing, but also various other areas relating to control flow. As it is based purely on messaging, it qualifies as a higher order mechanism for object oriented languages.

Using the curried argument message syntax, Higher Order Messaging is able to express collection processing operations without having to mention individual elements in the expression, unlike lambda based mechanisms. In addition, it is readily applicable to various other control flow applications, which can all be expressed as specialized message delivery mechanisms.

Higher Order Messaging is easy to implement, integrates well with existing languages and environments and has been proven to be reliable in demanding commercial applications.

More research is needed to look into reducing or even eliminating overheads due to the current implementation using message-failure hooks, for example by direct language support for sending Higher Order Messages. More direct support in the language for defining Higher Order Messages would also be helpful, possibly adapting the work done in functional languages for defining curried functions. Last but not least, there are probably many more applications of Higher Order Messaging waiting to be discovered.

9. REFERENCES

[1] BACKUS, J. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Commun. ACM 21*, 8 (1978), 613–641.

[2] BERGMANS, L. *Composing Concurrent Objects*. PhD thesis, University of Twente, June 1994.

[3] BROOKS, F. P. No silver bullet. *IEEE Computer 20*, 4 (Apr. 1987), 10–19.

[4] COX, B. *Object Oriented Programming, an Evolutionary Approach*. Addison Wesley, 1986.

[5] COX, B. J. Taskmaster ecoop position paper. In *ECOOP’91, Workshop on Exception Handling And OOPLS* (Geneva, Switzerland, 1991).

[6] D’HONDT, T., AND WOLFGANG. Of first-class methods and dynamic scope. In *Actes de LMO’2002: Langages et Modèles à Objets* (2002).

[7] DUCASSE, S. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP) 12*, 6 (June 1999), 39–44.

[8] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.

[9] GOLDBERG, A., AND ROBSON, D. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

[10] GOLDBERG, A., AND ROBSON, D. *Smalltalk-80: The Language*. Addison Wesley, 1989.

[11] IVERSON, K. E. *A Programming Language*. 1962.

[12] KÜHNE, T. Higher order objects in pure object-oriented languages. *SIGPLAN OOPS Mess. 6*, 1 (1995), 1–6.

[13] LIEBERHERR, K. J., SILVA-LEPE, I., AND XIAO, C. Adaptive object-oriented programming using graph-based customization. *Commun. ACM 37*, 5 (1994), 94–101.

[14] LISKOV, B., SNYDER, A., ATKINSON, R., AND SCHAFFERT, C. Abstraction mechanisms in clu. *Commun. ACM 20*, 8 (1977), 564–576.

[15] LUTZ, M. *Programming Python (2nd edition)*. O’Reilly & Associates, Inc., 1996.

[16] MATSUMOTO, Y. *The Ruby Programming Language*. Addison Wesley Professional, 2002. To appear.

[17] MCNAMARA, B., AND SMARAGDAKIS, Y. Functional programming in c++. In *ICFP ’00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming* (New York, NY, USA, 2000), ACM Press, pp. 118–129.

[18] MOUGIN, P., AND DUCASSE, S. Oopal: integrating array programming in object-oriented programming. In *OOPSLA ’03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications* (2003), ACM Press, pp. 65–77.

[19] MURER, S., OMOHUNDRO, S., STOUTAMIRE, D., AND SZYPERSKI, C. Iteration abstraction in sather. *ACM Trans. Program. Lang. Syst. 18*, 1 (1996), 1–15.

[20] MYERS, B. A., PANE, J. F., AND KO, A. Natural programming languages and environments. *Commun. ACM 47*, 9 (2004), 47–52.

[21] NAUGLER, D. R. Functional programming in java. *J. Comput. Small Coll. 18*, 6 (2003), 112–118.

[22] PANE, J. F. *A programming system for children that is designed for usability*. PhD thesis, 2002. Co-Chair-Brad A. Myers and Co-Chair-David Garlan.

[23] PRATIKAKIS, P., SPACCO, J., AND HICKS, M. Transparent proxies for java futures. In *OOPSLA ’04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications* (2004), ACM Press, pp. 206–223.

[24] ROBERT H. HALSTEAD, J. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst. 7*, 4 (1985), 501–538.

[25] WATANABE, T., AND YONEZAWA, A. Reflection in an object-oriented concurrent language. In *Proceedings OOPSLA ’88, ACM SIGPLAN Notices* (Nov. 1988), vol. 23, pp. 306–315.

[26] YOKOTE, Y. *The Design and Implementation of ConcurrentSmalltalk*, vol. 21 of *World Scientific Series in Computer Science*. World Scientific, 1990.