Don't Shoot the Messenger: A unified view of Senders and Receivers

Michael R. Levy
Department of Computer Science
University of Victoria
Victoria BC

Introduction

Although the idea of object-oriented programming are usually credited to the designers of Simula[1], it is the designers of Smalltalk[2] who are most responsible for the ideas considered today to be the core ideas of object-oriented programming. They could be considered revolutionaries: rather than adopt existing terminology (type, procedure, parameter passing and so on), they chose new words: object, class, message send and so on. A particularly elegant part of their philosophy was the attempt at the unifying idea that everything is an object. That is, even a class is an object. On closer inspection, it turns out that this is not quite true: in particular, messages are not objects (although their arguments are). Furthermore, mostly because of inheritance, classes are really the central concept of most object oriented programming languages.

We have recently developed an Internet scripting language called Woozle. Woozle was designed to be a kind of shell or scripting language (like Perl[3]), but for scripting on the Internet rather than on a single machine (or directory space). The starting point of the design of Woozle was that the central Internet activity is the transfer of one object to another (for example, sending an HTML file from a server to a Client). A typical Woozle script could be used to copy a sub-web (the April issue of Wired, for example) to a local machine. We adopted as the underlying design the idea that object-to-object transfer is naturally expressed as

receiver sender

where receiver denotes the receiving object and sender the object to be received\(^1\). We call this primitive operation composition. Contrast this with Smalltalk, for example, in which it is necessary to include a message keyword to achieve the same effect, as in, for example,

receiver get: sender

In Woozle, the meaning of composition is determined by the identity of the receiver and the type of the sender. In other words, message dispatch is determined by type matching rather than keyword selection. This idea turns out to have unexpected but pleasant consequences which will be discussed in this paper. As far as we know, this idea has not been explored before.

A Quick Overview of Woozle

Woozle is very much still in the design stage. There is an implementation (written in Java[4]) that can be run interactively, and which can used to download Web sites. However, we are still experimenting with important aspects of the language.

Woozle was motivated by the following problem. Suppose that you are the designer of a fairly complex Web site. Your documents may spread over several directories, and even, possibly, over several machines. Perhaps you designed your links so that the site is, in principle, easily moved from one machine to

\(^1\) This term is ambiguous: we mean the sent object rather than the agent that sent the object.
another. You may even wish to allow readers to download all or part of the site for off-line reading. How do you provide Web-site downloads?

A generic approach is to use a program that copies the Web site, given a root document. Although this is a simple solution, it has difficulties. Essentially, the problem is that a Web site is typically a sub-graph of a possible huge graph, and it is difficult for a generic program to know how to prune edges when making the copy.

A second approach is to allow the Web designer to specify the copy operation. They could, for example, provide a link like this: To download this months articles on widgets, click here. The link would download a script. The script would be run (using a helper program), and it would (a) retrieve the correct documents, and (b) recreate a copy of the remote directory structure on the local machine.

Woozle was designed specifically to allow this kind of scripting. The basic operation is object transfer (or, depending on your outlook, function composition). Thus, if \( L \) denotes a local object (for example, a file), and \( R \) a remote object (an html document, for example), then a "download" is written

\[
L \rightarrow R; \\
\text{and an "upload" is written}
\]

\[
R \rightarrow L; \\
\]

The implementation naturally takes care of details of the underlying protocol, issuing the appropriate GET or PUT requests and so on. At present, the operations are synchronous (so the program blocks until the transfer is complete), but we expect to develop an asynchronous version of the interpreter soon.

The Woozle interpreter supports integers, strings, lists and a few built-in objects, namely: \( \text{http, dir, file} \) and \( := \) (assignment). Expressions behave mostly as expected. For example,

\[
x := x + 1; \\
\]

will increment \( x \) by \( 1 \). The rule for a list receiver is this:

\[
[a_1, a_2, ..., a_n] [b_1, b_2, ..., b_m] = [a_1 b_1, a_2 b_2, ..., a_n b_m] \quad \text{where} \quad k = \min(n,m) \\
[a_1, a_2, ..., a_n] x = [a_1 x, a_2 x, ..., a_n x] \quad \text{for all other objects} \quad x
\]

Most objects iterate if the sender is a list:

\[
x [a_1, a_2, ..., a_n] = [x a_1, x a_2, ..., x a_n] \\
\]

These rules were designed to help specify concise down-loads. For example, suppose you wish to download some files to a local directory:

\[
\text{dir 'C:\mydir' (http://www.csr.uvic.ca/~mlevy/) ('f1', 'f2', 'f3', 'f4') 'html'}
\]

is a convenient short-hand for

\[
\text{dir 'C:\mydir' http://www.csr.uvic.ca/~mlevy/f1.html} \\
\text{dir 'C:\mydir' http://www.csr.uvic.ca/~mlevy/f2.html} \\
\text{dir 'C:\mydir' http://www.csr.uvic.ca/~mlevy/f3.html} \\
\text{dir 'C:\mydir' http://www.csr.uvic.ca/~mlevy/f4.html} \\
\]

Here is one more example. Suppose that your site contained a home page, two subdirectories, one for images and one for documents, and that you wished to download the home page, some of the images and two of the documents. The Woozle script might look like this:

\[
\text{local := dir 'C:\Wired\April'}; \\
\text{remote := http://www.wired.com/}; \\
\text{gifs := dir 'images' ['pic1.gif', 'pic2.gif', 'pic3.gif']}; \\
\text{docs := dir 'docs' ['doc1.html', 'doc2.html']}; \\
\text{local file 'index.html' remote '/'}; \\
\text{local dir 'images' gifs; } \\
\text{local dir 'docs' docs; } \\
\text{notify 'Download is complete'};
\]

Note that the objects dir, http, file, := and notify are Woozle built-in objects. Literals include lists, strings and integers. We have omitted function definition and the type system: these are still under development.
Our motivation for the omission of message keywords was conciseness - possibly a desirable goal for a scripting language. But the notion of a semantic variation for "Receiver Sender" based on the type of the senders seemed sufficiently rich to warrant further investigation. For example, in the current interpreter, String String means concatenation and File String means append a string to a file. It also turns out to be quite useful to allow a right-to-left association of objects. For example, suppose we wish to initialize several variables to zero. We can do this in the interpreter with

\[x, y, z := 0;\]

It seemed worth re-examining the basic notion of function application in light of the first Woozle interpreter.

**Objects and Primitive Arithmetic**

Smalltalk provides a coherent interpretation of conventional arithmetic expressions. An expression like

1+2*3

can be interpreted like this:

Send the message +2 to the object 1 (this will return the object 3)
Send the message *3 to the resulting object (3)

Note that the symbols "+" and "*" are not Smalltalk objects. They are special symbols used for method selection. Smalltalk (unlike Java) does support the passing of "functional" objects - they are called "blocks". For example, we could pass the + function to an object like this:

\[\text{obj} k:[i]i+i\]

but (a) we would need a keyword (k in this example) to denote the block argument, and (b) we need an explicit call (value) to evaluate the block.

Now let us start with 1+2*3 again, and consider the following interpretation:

Everything is an object.
Send + to the object 1. The result is the object (a function) \([x](1+x)\) (read this as "lambda x.1+x")
Send 2 to the object \([x](1+x)\). The result is the object 3.
Send * to the object 3. The result is the object \([x](3+x)\)
Finally, send 3 to the object \([x](3+x)\). The result is 9.

If "+" is an object, can it receive an argument? Yes it can. +2 (for example), evaluates to the object \([x](x+2)\)

where the \(\_\) is used to indicate that the function is a sender rather than a receiver (a right-side function, if you like). It is easy to show that

\[(x+)=x+\]

Suppose that \(L\) is a list of integers. What is the meaning of \(L\)? One reasonable interpretation is that \(x\) is sent to each element of \(L\). However, a nicer answer (and you will see why in a moment) is that the meaning should depend on the type of \(x\). If \(x\) is an arithmetic binary function, the send it to each element in the list. If \(x\) is a list, send each of its elements to the corresponding elements of \(L\), stopping (and truncating) at the shorter list. To summarize:

i) \([a_1,a_2,...,a_n]x = [a_1, x, a_2, x, ..., a_n, x]\) if \(x\) is of type \(n->n->n\)

ii) \([a_1,a_2,...,a_n][b_1,b_2,...,b_m] = [a_1, b_1, a_2, b_2, ..., a_n, b_m]\) where \(k = \min(m,n)\).

Although this interpretation arose from a desire to send remote HTML pages to local pages, it is quite pleasant for arithmetic purposes too. Here are two examples:

\[\begin{align*}
\{1,2,3,4\} + 2 &= \{1+2, 2+2, 3+2, 4+2\} = \{3,4,5,6\} \text{ by rule (i) and arithmetic.} \\
\{1,2,3,4\} + \{4,5,6,7\} &= \{1+(4),2+(5),3+(6),4+(7)\} \{4,5,6,7\} \text{ by rule (i)} \\
& \text{(recall that } n+ = [x](n+x) \text{ for any integer } n) \\
& = \{(1+4),(2+5),(3+6),(4+7)\} \text{ by rule (ii)} \\
& = \{5,7,9,11\}
\end{align*}\]

That is, the rules lead to point-wise application of binary operations to lists.
Our conclusion is that it is perfectly reasonable to treat higher-order functions as “objects”, that is, as entities that can be receivers or senders. The only trick is to extend (slightly) the usual notion of partial function application to support partial functions that must be senders (the usual notion, as in, for example, curried lambda expressions, is receivers). Thus (+1) for example, is an object which can be sent to an integer, just as (1+) is a receiver that is expecting an integer.

Conclusions

Woozle was originally designed to support Web scripting. The underlying premise of Woozle is that everything (including functions) should be an object. Woozle is similar in spirit to Self[5] that Smalltalk, since it has no intrinsic notion of class. Not only does it seem to be possible to provide coherent meaning to partial functions with arbitrary bound arguments, it is also practically convenient. In particular, flexible partial function application (combined with type-dispatching) enables a concise expression of Internet scripts.

Acknowledgments

Fang Lin implemented the Woozle interpreter.

References

2. Goldberg, Adele and David Robson. Smalltalk-80: The Language and Its Implementation, Addison-Wesley, Reading, MA, 1983.