ARITHMETIC AND DOUBLE DISPATCHING IN SMALLTALK-80

Kurt J. Hebel and Ralph E. Johnson[*]

University of Illinois at Urbana-Champaign

Object-oriented programming is ideal for many kinds of programming, such as simulation and graphics, and less suitable for other kinds, such as arithmetic. The problem with arithmetic is that in classical object-oriented languages, such as Smalltalk, all operations are messages sent to objects. An object responds to a message by looking in its class, finding a method (i.e. procedure) with the same name as the message, and executing it. Thus, the procedure that is invoked in response to a message depends only on the class of the receiver of the message, not of the argument. It seems illogical that adding an integer and a fraction is different from adding a fraction and an integer, but that is how Smalltalk works.

Some object-oriented languages, such as the Common Lisp Object System[1], determine a response to a message based on the classes of the arguments as well as of the receiver. While this is clearly more powerful than simple message sending, the extra power is usually not needed. Smalltalk's method lookup technique is sufficient for most operations. Still, there are many operations, especially arithmetic ones, that need to discriminate on the classes of several arguments.

One way to make an operation discriminate on the classes of several of its arguments is double dispatching, also called multiple polymorphism. Dan Ingalls first described how to use double dispatching in Smalltalk[5]. However, double dispatching is often criticized for being too complicated, for causing an explosion in the number of methods that have to be implemented, and for resulting in a system in which adding a new class requires changing existing classes. This paper shows that these problems are easily solved in Smalltalk-80 by adding a new tool to the programming environment, and that double dispatching is actually easier to understand, faster, and more flexible than alternative ways of implementing arithmetic in an object-oriented language. Moreover, double dispatching is a general technique and should be just as useful in other object-oriented programming languages.

---

[*] Address: Kurt Hebel, Kymatics, P.O. Box 2530, Station A, Champaign IL 61825-2530.

Ralph Johnson, Dept of Computer Science, 1304 West Springfield Ave, Urbana IL 61801 (217) 244-0093. email: johnson@cs.uiuc.edu

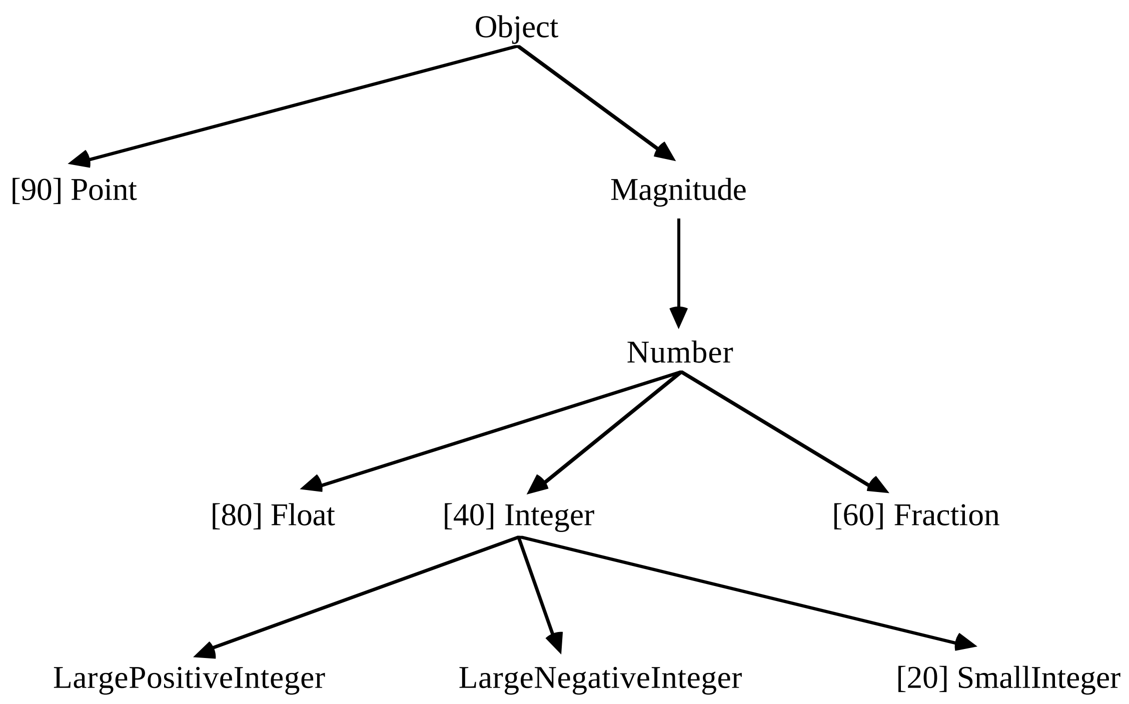


Figure 1. Part of the Smalltalk-80 class hierarchy. The number next to each class name indicates the generality of that class.

The Smalltalk-80 Arithmetic Classes

Smalltalk-80 has several different kinds of Numbers: Float, Fraction, SmallInteger, LargePositiveInteger, and LargeNegativeInteger. Class Point, which represents locations on the cartesian plane, can be added or multiplied by any of the Numbers, and so is also an arithmetic class. Figure 1 shows the portion of the class hierarchy that contains these classes.

Each arithmetic class in Smalltalk-80 has an integer-valued attribute called *generality*. The larger the generality of a class, the more general it is. An arithmetic class can represent all values of less general classes. Each class has a unique generality, so generality imposes a complete ordering on the arithmetic classes.

Generality seems similar to class inheritance, since more general classes include the values of the less general, but they are actually completely orthogonal. For example, even though every SmallInteger value can be represented as a Fraction, the representation of a SmallInteger is entirely different from that of a Fraction. A subclass inherits the representation of its superclass, so if SmallInteger were a subclass of Fraction then it would have at least as complicated a representation. Thus, even though Fraction is more general than SmallInteger, SmallInteger is not a subclass of Fraction.

Coercion

The arithmetic methods in Smalltalk-80 explicitly compare the classes of the receiver and the argument. If these classes are the same, the result is computed. If these classes are different, the retry:coercing: message is used to determine whether the receiver or the argument is more general, and *coerces* the less general Number to be in the same class as the more general Number. Once the receiver and the argument have the same class, the binary arithmetic message is *retried*, and the result of the retry is returned as the result of the original message send.

For example, evaluating ratio * 1.23, where ratio is a Fraction, causes the "*" method in Fraction to be invoked, which is

```
* aNumber
    (aNumber isMemberOf: Fraction)
        ifTrue: [
            ↑ (Fraction
                numerator: numerator * aNumber numerator
                denominator: denominator * aNumber denominator) reduced]
        ifFalse: [↑ self retry: #* coercing: aNumber]
```

The isMemberOf: method checks that the receiver is a member of the argument. Since 1.23 is a Float, it is not a member of Fraction, so retry:coercing: is sent. Its definition (which is inherited from Number) is:

```
retry: aSymbol coercing: aNumber
    (aSymbol == #= and: [(aNumber isKindOf: Number) == false])
        ifTrue: [↑ false].
    self generality < aNumber generality
        ifTrue: [↑ (aNumber coerce: self) perform: aSymbol with: aNumber.
    self generality > aNumber generality
        ifTrue: [↑ self perform: aSymbol with: (self coerce: aNumber)].
        self error: 'coercion attempt failed'
```

The first line checks whether the message being retried is "=". Since it is not, the method determines whether the receiver or the argument has greater generality, uses the coerce: message to coerce the object of lesser generality to the class of the object of greater generality, and retries the message using perform: with the new objects.

Although coercion can implement all the arithmetic classes in Smalltalk-80, it may not always work for other kinds of arithmetic classes. In particular, it does not work well for matrices. Multiplying a scalar by a matrix is different than coercing the scalar to a matrix and multiplying them: there are two kinds of multiplication for matrices. Double dispatching makes it easy to provide separate implementations for multiplying two matrices and for multiplying a scalar and a matrix.

Double Dispatching

Double dispatching is a way of choosing methods based on the class of the arguments as well as of the receiver. It involves a standard Smalltalk method–lookup, or message dispatch, for each argument, as well as for the receiver. Arithmetic operations usually have one argument, so there are two message dispatches — i.e. double dispatching. Dan Ingalls called this technique multiple polymorphism, which is more accurate but less poetic[5].

For example, the double-dispatch implementation of the method for "*" does nothing except send a message to the argument. The name of the message encodes the name of the original message (i.e. "*") and the class of the original receiver. The method for "*" in Fraction is

```
↑ aNumber productFromFraction: self.
```

Thus, multiplication of a Fraction and another number causes two message dispatches, one on "*" and one on productFromFraction:.

Each subclass of Number must implement the productFromFraction: method. These methods are usually very simple, since the class of the receiver and the argument are both known. Multiplying a Fraction with a Float will cause productFromFraction: to be sent to the Float with the Fraction as the argument. The definition of productFromFraction: in class Float is:.

```
productFromFraction: aNumber

        ↑ self productFromFloat: aNumber asFloat
```

which directly coerces the Fraction to be a Float. Multiplying two Floats is implemented by a primitive in Smalltalk-80, so the definition of productFromFloat: is

```
productFromFloat:  aFloat
    <primitive: 49>
    self error: 'The result cannot by represented by a Float.'.
```

In general, a double dispatch method for a message *op* is

argument *opWord*FromReceiverClass: self.

Since binary message selectors cannot be embedded in a keyword, if *op* is "+", "-", "*", "/", or "=" then *opWord* is "sum", "difference", "product", "quotient", or "equal". When double dispatching is used to implement regular keyword messages then *op* and *opWord* are the same.

Reducing the Number of Methods

An obvious disadvantage of double dispatching is that the number of methods can be very large. In the worst case, given n classes, there will be n^2 methods for each arithmetic operation. However, inheritance can be used in two different ways to reduce the number of methods. The most obvious is that similar classes such as Float and Integer can inherit methods such as **productFromMatrix:** from their common superclass, in this case Scalar.

The Smalltalk class hierarchy was changed to place all classes that can be used in arithmetic together under one abstract superclass, ArithmeticObject. This makes it possible to define very general versions of the double dispatching methods that can be inherited by most classes and overridden by a few. The new class hierarchy is shown in Figure 2.

The second way that inheritance can reduce the number of methods is to introduce inheritance along the class of the argument. For example, multiplication of a Matrix and a Float should not be any different than multiplication of a Matrix and an Integer, or, indeed, any kind of Scalar. This means that in class Matrix, the messages productFromFloat:, productFromInteger:, etc. perform the identical function. ArithmeticObject eliminates these redundant method definitions by providing a set of *forwarding* messages that implement productFromFloat: and productFromInteger: in terms of productFromScalar:. For example, the definition of productFromFloat: in ArithmeticObject is:

productFromFloat: aFloat

↑ self productFromNumber: aFloat

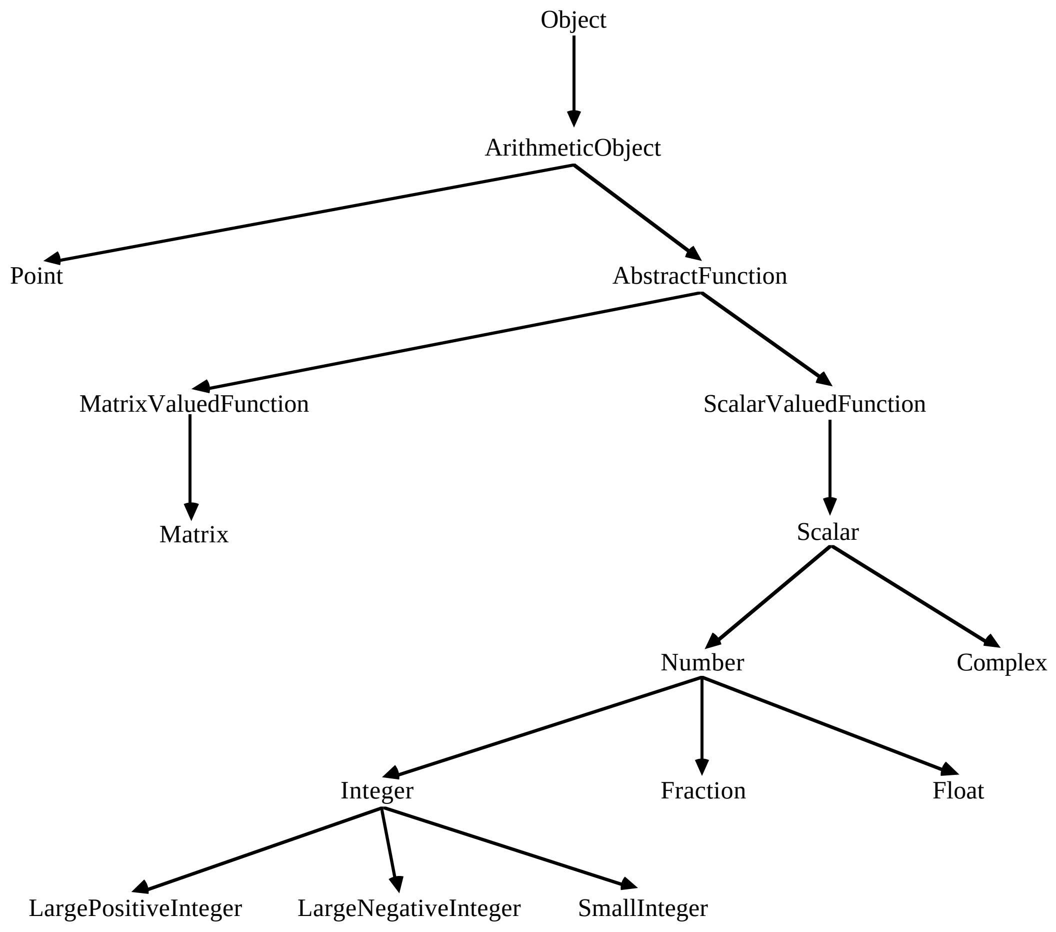


Figure 2. Part of the modified Smalltalk class hierarchy.

Thus, an arithmetic class that does not redefine **productFromFloat:** will use the same definition as for **productFromNumber:**. Messages from a class C get forwarded to a message from a superclass of C, so forwarding messages implement a form of inheritance along the class of the argument.

Forwarding messages allow the arithmetic operations to be defined only for the most general cases. For example, the definition of productFromNumber: in ArithmeticObject is

```
productFromNumber:  aNumber
    ↑ self productFromScalar: aFraction
```

The definition of productFromScalar: in ArithmeticObject is

```
productFromScalar:  aFraction
    ↑ self productFromScalarValuedFunction: aFraction
```

Class Matrix doesn't override any of these methods. Thus, multiplying any kind of number by a Matrix will result in sending the productFromScalarValuedFunction: message being sent to the Matrix. Its definition in Matrix is

```
productFromScalarValuedFunction:  aFraction
    | newMatrix |
    newMatrix  ← self copyEmpty.
    self withIndicesDo: [ :r :c :value |
        newMatrix at: r at: c put: aFraction * value].
    ↑ newMatrix
```

The Double-Dispatching Browser

One of the reasons that double dispatching seems complicated is that it is not supported very well by the standard Smalltalk-80 programming tools. These tools force the programmer to manually create all the methods required by double dispatching. Double dispatching methods fall into three categories:

1) those that send messages to arguments,

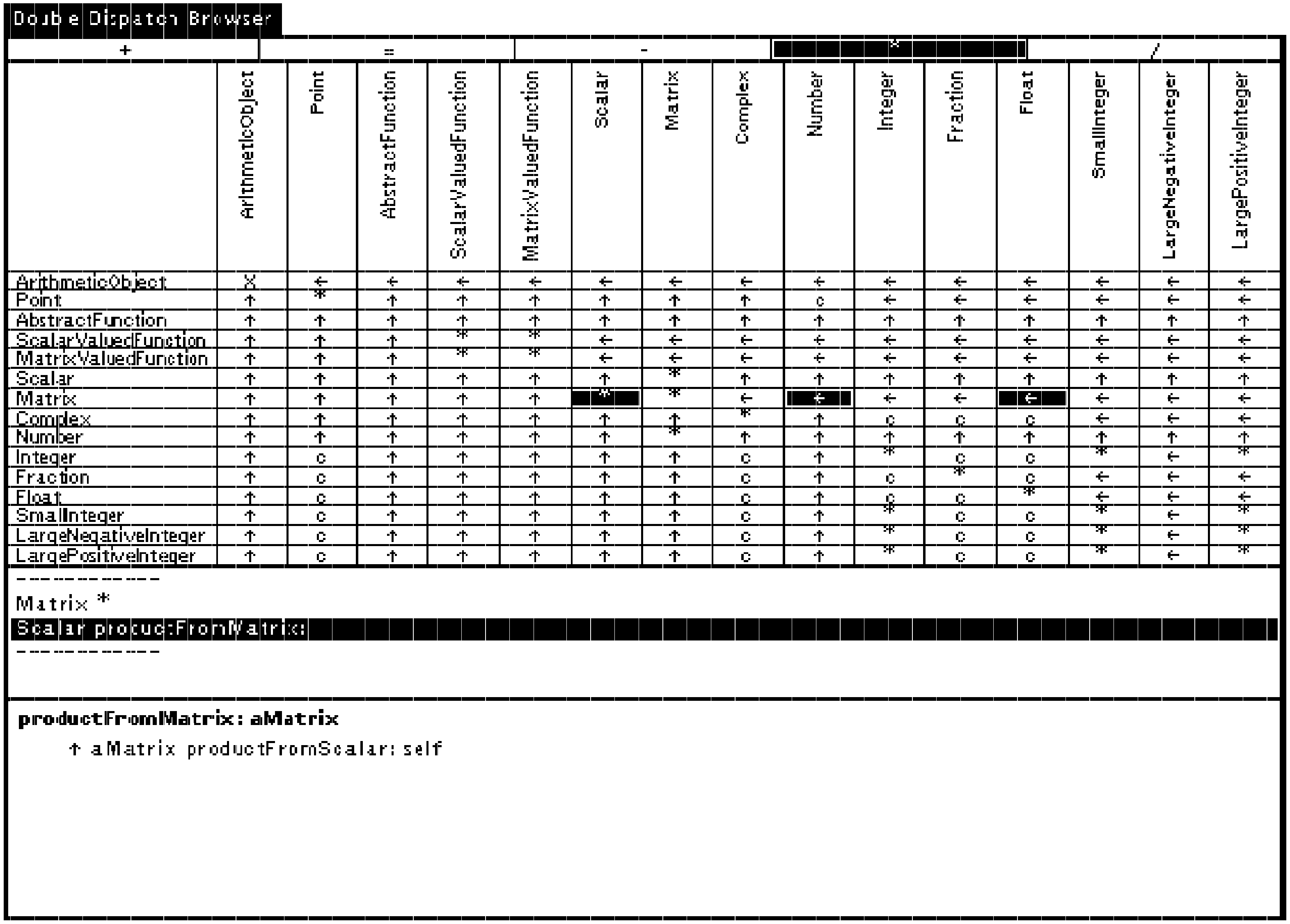
2) those that inherit from the superclass of the argument,

3) those final methods that get the work done.

We have developed a Double Dispatch Browser that automatically writes the first two kinds of methods and leaves stubs for the methods that do the work. Most of the methods are generated and installed automatically, making it much easier to use double dispatching. The Double Dispatching Browser is designed for methods that are implemented by double dispatching. The standard browser[2] should be used for other methods.

Figure 3 shows the Double Dispatch Browser. The top row indicates the set of message selectors that Arithmetic objects implement by double dispatching. The browser shows how each combination of classes responds to the selector. The two dimensional table in the center of the browser illustrates how the double dispatching methods are inherited. Each combination can either inherit from the receiver's superclass, from the argument's superclass, or can define a method for a particular combination of receiver and argument classes. The classes of the receiver run along the top, and the classes of the argument run along the left. A vertical arrow indicates inheritance from the superclass of the argument (forwarding), a left arrow indicates inheritance from the superclass of the receiver, an asterisk indicates that the method is defined in that class, a c indicates that the argument or the receiver is coerced to the other's class, and an X indicates an error. In the example shown, the user has selected Matrix * Float, which inherits from Matrix * Number, which inherits from Matrix * Scalar, which defines an implementation. The code view at the bottom of the browser displays the implementation for Matrix * Float, which is defined in Matrix * Scalar.

One advantage of the Double Dispatch Browser is that it provides a unified presentation of all implementations of a particular operation. The various methods for a single message, like “+”, can be thought of as fragments of the complete definition of “+”. The standard Smalltalk browser emphasizes classes and so makes it hard to think of how all the “+” methods are related to each other. The view provided by the Double Dispatch Browser is much more like that of the Common Lisp Object System, in which generic functions are thought to consist of a case statement that selects the right method to invoke. Both ways of thinking about classes and methods are useful, and the Double Dispatch Browser shows that they are compatible with each other. The Double Dispatch Browser is designed to be used in addition to the standard browser, not as a replacement for it.


Double Dispatch Browser

+ | = | - | × | /

	ArithmeticObject	Point	AbstractFunction	ScalarValuedFunction	MatrixValuedFunction	Scalar	Matrix	Complex	Number	Integer	Fraction	Float	SmallInteger	LargeNegativeInteger	LargePositiveInteger
ArithmeticObject	X	←	←	←	←	←	←	←	←	←	←	←	←	←	←
Point	↑	*	↑	↑	↑	↑	↑	↑	c	←	←	←	←	←	←
AbstractFunction	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑	↑
ScalarValuedFunction	↑	↑	↑	*	*	←	←	←	←	←	←	←	←	←	←
MatrixValuedFunction	↑	↑	↑	*	*	←	←	←	←	←	←	←	←	←	←
Scalar	↑	↑	↑	↑	↑	↑	*	↑	↑	↑	↑	↑	↑	↑	↑
Matrix	↑	↑	↑	↑	↑	*	*	←	←	←	←	←	←	←	←
Complex	↑	↑	↑	↑	↑	↑	↑	*	↑	c	c	c	←	←	←
Number	↑	↑	↑	↑	↑	↑	*	↑	↑	↑	↑	↑	↑	↑	↑
Integer	↑	c	↑	↑	↑	↑	↑	c	↑	*	c	c	*	←	*
Fraction	↑	c	↑	↑	↑	↑	↑	c	↑	c	*	c	←	←	←
Float	↑	c	↑	↑	↑	↑	↑	c	↑	c	c	*	←	←	←
SmallInteger	↑	c	↑	↑	↑	↑	↑	c	↑	*	c	c	*	←	*
LargeNegativeInteger	↑	c	↑	↑	↑	↑	↑	c	↑	*	c	c	*	←	*
LargePositiveInteger	↑	c	↑	↑	↑	↑	↑	c	↑	*	c	c	*	←	*

```
-------------
Matrix *
Scalar productFromMatrix:
-------------
```

```
productFromMatrix: aMatrix
    ↑ aMatrix productFromScalar: self
```

Figure 3. The Double Dispatch Browser. [Key to symbols: left and up arrows indicate inheritance and forwarding inheritance, X indicates an error, c indicates a coercion implementation, and * indicates a real implementation.]

Equality

There are two comparison messages that all Smalltalk objects must understand: identity ("==") and equality ("="). Identity has an intuitive meaning — the result of the message is true only if the receiver and the argument are the same object. Equality between two objects defaults — through inheritance — to the same meaning as identity, but is often redefined. In the case of Numbers, equality has been redefined so that two Numbers are equal if they have the same value, but not necessarily the same representation. Thus 0 (an Integer) and 0.0 (a Float) are equal but not identical.

In the original Smalltalk-80, equality was handled with coercion; that is, the less general Number was coerced to the more general Number and then retried, provided that the argument was some kind of Number. If the argument was not a Number, then the result was false.

Since any two objects may be compared for their equality, the equality methods for ArithmeticObjects must check the argument to make sure it is some kind of ArithmeticObject. For example, the definition of = in Fraction is:

```
= anObject
    anObject isArithmeticObject
        ifTrue: [↑ anObject equalFromFraction: self]
        ifFalse: [↑ false].
```

Without this check, all classes would need double dispatching methods, not just the arithmetic classes.

Type–Checking and Optimization

The original motivation behind rewriting the arithmetic classes was to make it easier to implement objects such as matrices. However, double dispatching has a number of other important advantages.

Typed Smalltalk is a derivative of Smalltalk-80 that includes type information[6]. It is being used as the basis of an optimizing compiler for Smalltalk that we are developing at the University of Illinois. Because case analysis of a message send is a fundamental part of Smalltalk, any successful type system for Smalltalk must support it. Since **retry:coercing:** is an ad–hoc technique that is not a fundamental part of the language, it is not supported very well. Rather than struggle with type–checking the retry:coercing: arithmetic of the original numeric classes, we use the implementation described in this paper. The kinds of optimizations performed by the compiler can make Smalltalk code using only Scalars, for example, as efficient as the equivalent C code. It is not clear whether we could have optimized the original code using coercion. Thus, double dispatching fits very well with compiled languages, as well as with standard implementations of Smalltalk.

One piece of evidence that double dispatching is simpler than coercion is the fact that it is easy to infer the types of the double dispatching methods[4], but it is hard even to type-check the original classes using coercion. Moreover, the Double Dispatching Browser can insert type information automatically.

Benchmarks

1) [fraction * float]]).
2) [float * fraction]]).
3) [fraction * point]]).
4) [point * fraction]]).
5) [fraction * fraction]]).
6) [float * float]]).
7) [point * point]]).

	1	2	3	4	5	6	7
Coercion:	(5259	5258	38396	33754	13929	1301	1630)
Dbl Dispatch:	(4059	3997	31765	31806	14134	1507	1671)
Ratio:	(77.1	76.0	82.7	94.2	101.5	115.8	102.5)

Figure 4. Relative performance of coercion and double dispatching. Times are in milliseconds for 10,000 operations.

Benchmarks to test the relative performance of coercion and double dispatching were run on a Macintosh II computer using ParcPlace Systems Smalltalk–80 version 2.3. Figure 4 shows the results of a few specific combinations of classes with multiplication. Coercion is a little faster than double dispatching when the operands are in the same class, but when the operands are in different classes then double dispatching is quite a bit faster.

The double dispatching is the same speed as coercion when all the operands are integers. This is because the interpreter does not perform a complete method-lookup for arithmetic operations when the receiver is a SmallInteger. (See pages 618-619 of Goldberg[3].) Thus, addition and multiplication of SmallIntegers does not depend on their definitions in the class hierarchy, but only on their definition in the interpreter. The double dispatching implementation of addition and multiplication of Floats would be as fast as the original coercing implementation if the primitive definition were in the the first method that responds to the operation instead of in the second method, because primitives fail when their arguments are of the wrong type. However, this technique can only be used for methods defined by primitives so we did not to use it in this benchmark.

Figure 5 shows the number of methods for coercion, double dispatching without method reduction (assuming n^2 methods), and double dispatching with method reduction. Only the basic messages need be written by the programmer, all other can be inserted by the double dispatch browser.

	Coercion	Double Dispatching without Reduction	Double Dispatching with Reduction
Basic Messages	180	180	180
Coercion Support	73	0	0
Double Dispatch Support	0	6480	221
Totals	253	6660	401

Figure 5. The number of methods required to implement the complete arithmetic class hierarchy (36 classes and 5 operators).

Conclusions

Double dispatching is much more flexible than coercion and is easier to understand. Well-designed tools can minimize the number of methods that the programmer must think about. Double dispatching is faster when performing arithmetic between different classes, but is a little slower when the operands are in the same class. It is probably even better suited for compiled languages than for interpreted languages, but it is currently our preferred technique for implementing arithmetic in Smalltalk–80.

We have developed a complete set of arithmetic classes using double dispatching that includes complex numbers, extended precision floating point numbers, matrices, and functions as arithmetic objects. Some of these kinds of objects, like extended precision floating point numbers, could have been implemented just as easily with coercion. Others, like matrices, could not have been implemented with coercion without a great deal of effort. They all fit into the double dispatching framework very nicely.

For more information on the Double Dispatching Browser and on the arithmetic classes, contact the first author.

Acknowledgements:

This work was supported in part by NSF contract CCR-8715752 and by the AT&T grant to the Illinois Software Engineering Program. Many thanks to Justin Graver for type-checking the double dispatching arithmetic classes. We also thank Simon Kaplan and Brian Marick for helpful comments on earlier drafts of this paper.

References:

[1] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp Object System Specification. Printed as SIGPLAN Notices, Special issue, September 1988.

[2] Adele Goldberg. *Smalltalk–80: The Interactive Programming Environment*. Addison–Wesley, Reading, Massachusetts, 1984.

[3] Adele Goldberg and David Robson. *Smalltalk–80: The Language and its Implementation*. Addison–Wesley, Reading, Massachusetts, 1983.

[4] Justin O. Graver. Type-Checking and Typed-Inference for Object-Oriented Programming Languages. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1989.

[5] Daniel H. H. Ingalls. A Simple Technique for Handling Multiple Polymorphism. In *Proceedings of OOPSLA '86, Object–Oriented Programming Systems, Languages and Applications*, pages 347–349, November 1986. Printed as SIGPLAN Notices, 21(11).

[6] Ralph E. Johnson, Justin O. Graver and Larry W. Zurawski. TS: an Optimizing Compiler for Smalltalk. In *Proceedings of OOPSLA '88, Object–Oriented Programming Systems, Languages and Applications*, pages 18–26, November 1988. Printed as SIGPLAN Notices, 23(11).