Generic Operators
A Way to Build Flexible Systems

João Victor Lopez Pereira

November 25, 2024

Rio de Janeiro - RJ

Abstract

This document explores the concept of Generic Operators in programming, focusing on the flexibility and extensibility they offer by employing Data-directed Programming techniques. It highlights the problem of "dead end" programs — those that require significant rewrites when new requirements arise — and proposes a solution using a dynamically typed language to manage different data types efficiently. By leveraging tagged data and a table of operations, the system avoids cumbersome type checks and scales gracefully with additional data representations.

The document provides an example of how to implement arithmetic operations for various numerical types (integers, rationals, reals, and complex numbers) without needing to manually handle each type combination. It introduces procedures for managing operations dynamically and demonstrates how this approach can be extended to include new data types, such as polynomials and matrices, with minimal additional code. The result is a highly flexible arithmetic system that supports a wide range of data structures and operations, emphasizing the power of Generic Operators and Data-directed Programming in creating adaptable, flexible and robust systems.

Generic Operators

> It seems to me that one of the biggest problems people have with programs is writing programs that are dead ends. What I mean by dead end is: you've written this big complicated piece of software and then the world changes and something else is needed for it to be done and then you have to rewrite out a big chunk of it.
>
> —Gerald Jay Sussman,
> *"Flexible Systems, The Power of Generic Operations"*[4]

Generic Operators are procedures capable of operating on multiple data types. dynamically typed languages make it convenient for the user to consider adding a tag to an object before calling a function to operate on it. For example:

```
(define (type? x)
  (cond ((string? x)
         (do-something-1))

        ((number? x)
         (do-something-2))

        ((char? x)
         (do-something-3))

        (else
         (do-another-thing))))
```

However, the problem with this system is that a "manager" is needed to keep incrementing all the checks in any function that might use all the types of data that are being manipulated. We can use a technique called Data-directed Programming to solve this problem. Data-directed Programming is a programming paradigm in which the data structures that are being manipulated carry information on how to operate on them. For example: in a system like the one above, the types `string`, `number` and `char` could be implemented with a tag, in a way that a `string` is in reality a pair containing a tag and the string itself. The process of assigning labels to provide a better manipulation of data structures is called Data Tagging, and these objects can be referred to as tagged data.

If `string` is in reality a pair, we can easily implement these type predicates:

```
(define (string? x)
  (eq? (car x) 'string))

(define (number? x)
  (eq? (car x) 'number))

(define (char? x)
  (eq? (car x) 'char))
```

Even though we have a nice way of dealing with the dynamic typing of the language, we haven't really dealt with the manager. Now, suppose we want to implement an arithmetic system. Let's focus only on the procedure responsible for adding two numbers. Assuming the primitive `+` does not operate on multiple data representations, we would need to implement a way to make the system check which type we are dealing with. Something like:

```
(define (no-no-add x y)
  (cond ((and (integer? x)
              (integer? y))
         (add-int x y))

        ((and (rational? x)
              (rational? y))
         (add-rational x y))

        ((and (real? x)
              (real? y))
         (add-real x y))

        ((and (complex? x)
              (complex? y))
         (add-complex x y))

        (else "this procedure can't deal with these representations")))
```

This procedure needs to check for each of our representations. That means that if we name n types in our system, we would need n checks in each of our procedures that manipulate all of them. You can see as well that we cannot add two numbers that have a different representation. If we wanted to do so, the complexity of our system would skyrocket. If we have n representations and *add* is a binary system, we would need approximately n^2 checks in each of our procedures that deal with all of our data structures. There are some ways of doing less than n^2. For example, the programmer can somehow check if the representations are the same in a single line. That way he would have $n^2 - n$ checks, which doesn't really make a difference. This could be easily implemented with the tagged data technique.

However, we are interested in building dynamic and flexible systems. We don't want to worry about implementing plenty of functions every time a new representation is added.

> It's really annoying that the bottleneck in this system, the thing that is preventing flexibility and change, is completely in the bureaucracy.
>
> —Harold Abelson,
> *"Lecture 4B: Generic Operators"*[1]

Well, a solution for this are 2 procedures:

- one that stores a given value in a table
- and one that retrieves a value from a given table

Doing so, we can combine both techniques to not deal with all the checks. Let's call the first procedure `put!` and the second one `get`.

```
(put!  key1 key2 value)
```

```
(get key1 key2)
```

We could have built these procedures to simply use one key, but since we have a system with multiple operations and multiple operands, it's better if we have one specific key for each one of them.

First, we need to implement the add procedures used in the `no-no-add` definition above. These implementations are simple, and once we know how one of them is implemented, the others become trivial.

```
(define contents cdr)

(define attach-tag cons)

(define (make-int x)
  (attach-tag 'Z x))

(define (add-int x y)
  (make-int (+ (contents x)
               (contents y))))
```

From this piece of code, we can see that our integer representation is simply an integer attached to a tag, which is the symbol `Z` to represent the mathematical notation for the integer numbers set. Just as in math, we can create a representation of rational numbers as two integers:

```
(make-rat (make-int 5) (make-int 3)) ;; --> ('Q (('Z 5) ('Z 3)))
```

In the **generic-operators.scm** file, the precise definition of the functions used here can be found. The objetive of this document is not to dive into detail, but to justify that such a radical approach as getting rid of the "manager" makes up for a flexible and extendable system.

We can represent a real number in a very similar way as we represent an integer and we can represent a complex number a similar way as we represent a rational:

```
(make-real 5.2) ;; ~~> ('R 5.2)

(make-complex (make-real 4.2) (make-real 2.1)) ;; --> ('C (('R 4.2) ('R 2.1)))
```

After implementing these, we can add them is the table[1]:

```
(put! 'add 'int add-int)
(put! 'add 'rat add-rat)
(put! 'add 'real add-real)
(put! 'add 'complex add-complex)
```

We can easily see that we have a sort of data hierarchy going on. Considering that mathematically a rational number is defined using integer numbers and a complex number is defined using real numbers, we have a structure here that is called a numerical tower.

Mathematically, the set of complex numbers includes the set of real numbers, which in turn includes the set of rational numbers, and this set includes the set of integers. This hierarchy allows us to implement a flexible approach to operate on different types, meaning that we can add two numbers from different sets, as one set is a subset of the other. We can create a procedure that maintains cohesion and converts a number into different representations. For example, an integer can be treated as a rational number with 1 as its denominator. Similarly, a real number can be derived from a rational number by dividing the numerator by the denominator. Additionally, a real number can be viewed as a complex number with its imaginary part equal to 0. If we implement this procedure that does a `integer->rational` conversion and we have a `rational->real` conversion, we, by the transitivity rule, can automatically have a `integer->real` procedure. This way, we have $n - 1$ functions left to implement.

We can also create a way to simplify a number if its representation is unnecessarily complicated. For example, we can treat the complex number $2.0 + 0i$ as the integer 2. We would have to implement $n - 1$ functions just like we created to "raise" our representation.[2] This way, we have $2n - 2$ procedures, which is a lot better can n^2.

The reader might be thinking a way to implement this functions or taking a step further and thinking about how to implement all the checks without the "manager". We introduced two procedures: `put!` and `get`, and in their power lays the answer.

Combining all the techniques we mentioned until this moment, we can implement a dynamic system that, by the use of a list-like table, tagged data, data directed programming and these cohesion procedures, is able to do something like:

[1] We previously said that `put!` stores a value in a table. Since we are using Scheme and Scheme supports Higher Order Procedures, we treat functions as first-class elements.

[2] The procedure `real->rational` is tricky. How can we check if a real number can be treated as a rational one? We decided not to go down the rabbit hole.

```
(define (add x y)
  (if (eq? (tag x) (tag y))
      (simplify ((get 'add (tag x)) x y))
      (let ((a (tag->number (tag x)))
            (b (tag->number (tag y))))
        (if (a < b)
            (add (raise x) y)
            (add x (raise y))))))
```

This is not exactly the way we implemented this procedure, however; it's a much simple way that represents with a certain level of correctness the logical idea behind our implementation. The procedures checks recursively if a data type should be "raised" into a set that containing the set that the number is currently in. In the other words, the procedure will continue changing the representation of our data structures until they have the same tag, i.e., they are both of the same numerical type.

In reality, our add procedure is implemented as a `fold-left` of a more generic procedure that returns the lambda that helps me create add, and it makes it so far easier to implement all operators.

Just like that, we have a system that can deal with complex numbers, real numbers, rational numbers and integer numbers and we haven't used a single check to see if a data structure has a specific type. The procedures to change the representation of a data structure and be defined by using the table as well as the keys stored in the tagged datas to find the procedure to modify it accordingly. The real question is: what happens if we would like to implement another representation into our system?

Suppose we want to add polynomials. First, we have got to choose a representation for them. Given a representation, the procedures that manipulate them are obvious. In this specific implementation, we decided that a polynomial is a pair containing its tag and another containing more information, such as the variable the polynomial is defined in and its terms.

```
('polynomial ('var ((c1 e1) (c2 e2) (c3 e3)...)))
```

Given a representation, we have to create the procedure responsible for adding polynomials. Only its most important part will be showed since the whole procedure is not interesting at this level of detail we are in.

```
(define (add-polynomial x y)
  <...>
  (cond ((null? x) y)
        ((null? y) x)
        ((< (contents (exp x)) (contents (exp y)))
         (cons (term x) (iter (remaining-terms x) y)))
        ((< (contents (exp y)) (contents (exp x)))
         (cons (term y) (iter x (remaining-terms y))))
        ((= (contents (exp x)) (contents (exp y)))
         (cons (make-term (+ (coeff x) (coeff y))
                          (exp x))
               (iter (remaining-terms x) (remaining-terms y))))))
```

```
    <...>)
```

After having the procedure created, we can simply add it in the table:

```
(put! 'add 'polynomial add-polynomial)
```

By doing this, there are things from which that we need to be careful:

- The polynomial isn't part of our numerical tower, i. e. it cannot be added with any other thing that isn't a polynomial of the same variable;
- The polynomial doesn't use our data structures as its coefficients.

The first observation is far more complicated to deal with than the second. In fact, the second one is dealt only by changing 3 characters in the `add-polynomial` definition. Instead of using the native `+` procedure, we can substitute it for `add`, and doing so, now we can add polynomials that have complex numbers, real numbers, rational numbers and integer numbers as its coefficients since all of these are added using `add`. However, a polynomial itself can be added by the procedure `add`. What it means is that the simple fact that we wanted our polynomials to be able to have our data structures as its coefficients resulted in a recursive definition of `add` that can be heavily used in our favour.

```
(define (add-polynomial x y)
  <...>
  (cond <...>
        ((= (contents (exp x)) (contents (exp y)))
         (cons (make-term (add (coeff x) (coeff y))
                          (exp x))
               (iter (remaining-terms x) (remaining-terms y)))))
    <...>)
```

By doing so, we have all four numerical operands defined in terms of `+` and the polynomial type defined in terms of `add`. This recursive definition is finite since a polynomial is made of finite parts that have finite coefficients, i.e. the recursive process of adding polynomials with polynomials as coefficients will come to an end since they aren't infinite.

Suppose we want to have a representation for a matrix. Let a matrix be a list made of lists that same the same length.

```
    ('matrix ((a11 a21 a31...)  (a12 a22 a32...)  (a13 a23 a33...)  ...))
```

Having decided a representation, we can implement a way of adding matrices.

```
(define (add-matrix m1 m2)
  (define (add-row row1 row2)
    (map add row1 row2))

  (define (iter rows1 rows2)
```

```
    (cond ((null? rows1) '())
          ((null? rows2) (error "Matrices have different dimensions"))
          (else (cons (add-row (car rows1) (car rows2))
                      (iter (cdr rows1) (cdr rows2))))))

  (if (same-dimension? (contents m1) (contents m2))
      (cons 'M (iter (contents m1) (contents m2)))
      (error "matrices do not have the same dimension")))

(put! 'add 'matrix add-matrix)
```

If the matrices that the same dimension and have terms left to be added, than we call `add-row`. What this procedure does is use `map` to add all the terms of two given vectors. If we define `add-row` using `add` and insert `add-matrix` in our table, then we have the capacity of implementing matrices made of polynomials that can be made of polynomials, complex numbers, real numbers, rational numbers and integer numbers. This way, we can have this as one of the things we are able to add.

$$\begin{pmatrix} 5x^0 + \frac{1}{2}x^2 + (5.2 + 4.6i)x^5 & 2.4x^2 + (4y^1 + 3.5y^3)x^3 \\ (2.3z^1 + 3z^4)x^1 + \frac{3}{4}x^2 & 6x^0 + (4.3 + 2.0i)x^3 \end{pmatrix}$$

This way, we have implemented an arithmetic system that is capable of dealing with tons of representations and making use of a extremely flexible and dynamic system that can be extended only by a single call to the `put!` procedure, since the operators will automatically know how to deal with that representation. What we are doing is using a Generic Operator `add` that is capable of making me define these different data structures and make one be part of the others by using the generic `add` procedure.

This way, we've built a system that has decentralized control: no manager, no n^2 necessary procedures to manipulate the representations and no n^2 procedures to coerce representations into the correct type before adding. The time complexity of our system isn't the best it can be. We are not interested in the computational effort that will be necessary to add numbers, what we are really worried about is the power of representation, flexibility and extensibility that the techniques we are using provide to the programmer when implemented in a technique that is pretty much appropriated for dealing with Data-directed programming, data hierarchy, recursion and tagged data.

This system wasn't made to substitute an existent arithmetic calculator, but only to show the power that all these techniques and programming paradigms have when used properly.

> Every program has (at least) two purposes: the one for which it was written, and another for which it wasn't.
>
> —Alan Jay Perlis,
> *"Epigrams on Programming"*[3]

Bibliography

[1] Harold Abelson. *Lecture 4B: Generic Operators*. Accessed August 7, 2024. Timestamp: 29:05. URL: https://youtu.be/OscT4N2qq7o?si=6qc8ZRfTzYxZ4Bi2&t=1744.

[2] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. Cambridge, Massachusetts: The MIT Press, July 1984. ISBN: 9780262010771.

[3] A. J. Perlis. "Epigrams on Programming". In: *ACM SIGPLAN Notices* 17.9 (Sept. 1982), pp. 7–13. DOI: 10.1145/947955.1083808. URL: https://web.archive.org/web/19990117034445/http://www-pu.informatik.uni-tuebingen.de/users/klaeren/epigrams.html.

[4] Gerald Jay Sussman. *Flexible Systems, The Power of Generic Operations*. Timestamp 2:20. URL: https://www.youtube.com/watch?v=cblhgNUoX9M.