Higher order programming using *Mathematica* in teaching programming

Hiroshi Ohtsuka
Graduate School of Mathematics
Kyushu University
ohtsuka@math.kyushu-u.ac.jp

Abstract

In teaching programming for students in mathematics course, one of the important features for programming languages may be the ability to treat functions as "*first–order*" objects. *Mathematica* provides this facility as *higher order functions*.

In this paper, we shall use examples to present not only advantages of this feature by comparison with *C* but also address problems arising from the "*type–freeness*" of *Mathematica*, which can be partially analyzed and resolved by "*meta–programming*" in *Mathematica*.

1 Introduction

This paper was triggered by two motivations in my experience of teaching algorithms with exercises for their programming to senior students in mathematics course in the first term at Kyushu University.

1. They have been interested in using function arguments, i.e., pointers to functions, to program algorithms such as numerical integrations.

2. We have often used function arguments in measuring computational times and other properties of algorithms to analyze and compare them each other.

It is preferable for us to handle functions as "first–order" objects. That is, functions can be passed as arguments to other functions and can be handled as data, e.g., can be included in any data structures.

Mathematica provides this facility as higher order functions, which take functions as arguments or produce functions as results, and as no distinction between functions and data, both of which are uniformly represented as *expressions*. In particular, *Currying*, which enables a function to be *partially applied* to form a new function, characterizes *Mathematica* as a higher order functional programming language.

Before I adopted *Mathematica*, I had used C and C++ to teach programming for students who are almost beginners for the computer. Though these languages provide the facility of function argument, which is a restricted form of higher order functions, it was impossible for us to teach those students function arguments systematically only in the half term. The main reason for this is that these languages require us to grasp their type systems. Therefore, I had taught them function arguments a priori when necessary. On the other hand, *Mathematica* does not require such preliminaries and, in fact, we could proceed to the exercises. There may be a further important reason why *Mathematica* expedited our exercises: students could recognize function spaces mathematically so they might be able to handle higher order functions by the analogy with it.

This paper is organized as follows. In section 2, we take a glance at mathematical background of higher order functions. In section 3 and 4, we present the facility of higher order functions, their advantages compared to C and some standard ones working over lists with examples. In section 5, we turn our attention to a problem arising from the relation between higher order functions and type–freeness of *Mathematica*, which can be partially analyzed and resolved by "meta–programming" in *Mathematica*. The last section concludes with some results of student questionnaires about higher order functions.

2 λ–calculus and Cartesian Closed Categories

In this section, we briefly look at mathematical backgrounds of higher order functions, known as λ–*calculus*[1] and *cartesian closed categories*[2].

λ–calculus is used to describe the semantics of functional languages. In λ–calculus, λ–*terms* are built from *variables* x, y, ... by *application* (MN) (M`[N]` in *Mathematica*) and *abstraction* $(\lambda x.M)$ (`Function[`x`,`M`]` in *Mathematica*), where M, N are λ–terms. Parentheses will be omitted if no confusion occurs.

The Computation (reduction) in λ–calculus consists of repeatedly performing β–reduction according to the rule:

$$(\lambda x.M)N \longrightarrow M[x := N] \tag{1}$$

where R.H.S. denotes the result of substituting N for the free occurrences of the variable x in M without variable conflict (known as α–conversion). We do not go into further detail about λ–calculus.

In the L.H.S. of (1), abstraction turns M (involving the variable x) into the function $\lambda x.M$. Then L.H.S. itself can be regarded as the function application of $\lambda x.M$ with parameter x to the argument N, and contraction results in R.H.S. The following *Mathematica* session describes this situation.

```
In[1]:= Function[x, t[x]][y]
Out[1]= t[y]
In[2]:= Function[x, Function[y, f[x, y]]][y]
Out[2]= Function[y$, f[y, y$]]
In[3]:= Function[y, f[x, y]] /. x -> y
Out[3]= Function[y, f[y, y]]
```

We remark that `Out[2]`, the result of the substitution, is different from `Out[3]`, the result of the replacement, as `Out[2]` indicates the necessity of an α–conversion in the former.

Cartesian closed categories abbreviated to CCC are categories with products and exponents. They are also used to describe the semantics not only of functional languages but also of λ–calculus[3]. In CCC, for an arrow with a product as its domain $f : A \times B \longrightarrow C$, there exists unique arrow with the exponent range $f^* : A \longrightarrow C^B$ which satisfies:

$$f = \varepsilon < f^*\pi, \pi' > : A \times B \longrightarrow C^B \times B \longrightarrow C \tag{2}$$

where π and π' are the 1st and 2nd projections respectively, ε is the evaluation arrow. We do not go into further detail about CCC.

Equation (2) can be seen as $f(x, y) = \varepsilon(f^*(x), y) (= f^*(x)(y))$ for any $(x, y) \in A \times B$. In CCC, the abstraction of a λ–term may be seen as making f^* from f (a partially applied form $\lambda y.f(x, y)$ from $f(x, y)$) which amounts to *Currying* in functional programming. The application of a λ–term to another one is replaced by the composition of ε to their pair. The following *Mathematica* session describes this situation.

```
In[4]:= Function[z, f[x, z]][y]
Out[4]= f[x, y]
In[5]:= Apply[Function[z, f[x, z]], {y}]
Out[5]= f[x, y]
```

We remark that as `Apply` replaces the head of its 2nd argument by its 1st argument, we need to wrap 2nd argument with `List` as in `In[5]`.

3 Examples of Higher Order Functions

Before we demonstrate the ability of higher order functions we briefly look at the syntax of *Mathematica*. Expressions in *Mathematica* can be written uniformly in the form h`[`e_1`,` e_2`,` $\cdots$`]` constructed from atomic objects. h is called the *head* and the e_i are called the *elements*. Both head and the elements may themselves be expressions. It is obvious that the syntax of *Mathematica* is far simpler than that of C so it is easy to learn.

Many functions can be made more general by substituting explicit function calls with parameters. The decision as to which function is actually employed is deferred until the function is applied.

3.1 Numerical integrations

In ATCM97, Wei–Chi Yang [4] demonstrated numerical integrations using *Maple*. He introduced left sum $L_f(a,b,n)$, right sum $R_f(a,b,n)$, trapezoidal sum $T_f(a,b,n)$, midpoint sum $M_f(a,b,n)$ and Simpson sum $S_f(a,b,n)$ of a function f over n subintervals of $[a,\ b]$ to approximate $\int_a^b f(x)dx$. We show the same examples from different point of view and with comparison to C.

Using function arguments, these sums are defined in C as follows. First, we prepare a general function taking an integrand and an area of subinterval as arguments.

```
double integral(double (*func)(double), double a, double b,
     double (*sum_func)(double (*)(double), double, double), int n)
{
  double sum = 0.0, delta = (b−a)/n, left;
  int i;
  for(i = 0, left = a; i < n; i++, left += delta)
    sum += (*sum_func)(func, left, left + delta);
  return sum; }
```

Then, it is sufficient to define the functions for areas of subinterval and for approximations corresponding to $L_f(a,b,n)$, $\cdots$. It is important that we do not need to prepare them separately.

```
double left(double (*func)(double), double a, double b) {
  return (*func)(a) * (b−a); }
/* correspond to L_f(a,b,n) */
double left_sum(double (*func)(double), double a, double b, int n) {
  return integral(func, a, b, left, n); }
      ⋮
```

Although function arguments in C may often help to simplify code and so lead to concise programs that are more general, it is almost impossible for students to accomplish them in only the half term. The main reason for this is that they needed to understand the pointer and its relation to other types in C, such as priority. Therefore, I had taught them function arguments a priori when necessary.

On the other hand, they can write the following functions in *Mathematica*, equivalent to the above C functions, without any of the above preliminaries.

```
integral[func_, a_Real, b_Real, sumfunc_, n_Integer] :=
  Module[{sum = 0.0, delta = (b−a)/n, left = a, i},
          Do[sum + = sumfunc[func, left, left + delta]; left += delta,
            {i, 1, n}];
```

```
        sum]

left[func_, a_Real, b_Real] := func[a] * (b−a)
    ⋮
```

I guess that the pointer in C disturbs their understanding of functions arguments though they can understand function spaces mathematically and may be able to handle higher order functions by the analogy with it.

From the higher order functional point of view, the reason why C has only restricted forms of higher order functions is that we must define functions such as `left_sum` as it is impossible to handle partially applied functions as data. On the other hand, *Mathematica* can do it, e.g., we can include `integral[func,a,b,left,n]` in which all arguments except `left` are formal parameters in any expressions. See the function `compare` in section 4.

3.2 Sorting algorithms

The next example is sorting which is well–known by computer science students but known by few mathematics students. In [5] Maeder supplied insertion, selection and quick sorts. We adopt and generalize the bubble sort so that it will sort a list of any type in either ascending or descending order.

```
BubbleSort[list_List] :=
  Module[{l = list, i, j, n = Length[list]},
    Do[Do[If[Greater[l[[j−1]], l[[j]]], {l[[j−1]], l[[j]]} = {l[[j]], l[[j−1]]}],
          {j, n, i, −1}],(* end of inner Do *)
       {i, 1, n−1}];      (* end of outer Do *)
    l]                    (* return l *)
```

The only alternation required to `BubbleSort` is to substitute a comparison function in place of *Greater* in it.

```
BubbleSort[list_List, Order_] :=
  Module[{l = list, i, j, n = Length[list]},
    Do[Do[If[Order[l[[j−1]], l[[j]]], {l[[j−1]], l[[j]]} = {l[[j]], l[[j−1]]}],
          {j, n, i, −1}],
       {i, 1, n−1}];
    l]
```

In this manner, the functionality of `BubbleSort` has been increased significantly since a different *Order* can be slotted in place by a simple declaration, not restricted to `Greater` and `Less` for numbers, but also user–defined `StringLess`, `StringGreater` for strings or `ExprLess`, `ExprGreater` for arbitrary expressions. In the same manner, we can extend `InsertionSort`, `SelectionSort` and `QuickSort` in [5].

4 Higher order functions over lists

It is very hard for students in mathematics course to handle lists or any *Mathematica* recursive expressions using recursions over them. Several powerful higher order functions on expressions are useful for students because they avoid the use of explicit recursion, thereby not only producing programs which are shorter, easier to read and to understand but also eliminating errors such as infinite loops. Their functionalities are quite different from those introduced in Section 3 and most of them are included in the following three well known families in functional programming:

- The '`Map`' family which retains the list structure but transforms the list items, so that it returns the list of the consequences. `Scan`, `Thread`, `MapThread` and `Through` are in this family.
- The '`Fold`' family which distributes a dyadic function over a list, so that it works over a list, typically producing single value or a list of values. This family contains `Nest`, `NestList`, `FoldList` and `FixedPoint`.
- The '`Select`' family which retains the list structure but may delete items from the list, according to a given predicate. This family has `Cases`.

There are many other useful higher order functions, such as `Distribute`, `Inner` and `Outer`.

Next, we show an example of the most–commonly used higher order function `Map` (though more complicated ones may be able to simplify it further). If we want to compare the accuracies of approximations for the numerical integrations of a function over an interval, we can write the following function.

```
compare[func_, a_, b_] :=
  Module[{steps = Range[4, 100, 4]},
    Print["left sum:  ", Map[integral[func, a, b, left, #]&, steps]];
    Print["right sum:  ", Map[integral[func, a, b, right, #]&, steps]];
    Print["trapezoidal sum:  ",
          Map[integral[func, a, b, trapezoid, #]&, steps]];
    Print["midpoint sum:  ",
          Map[integral[func, a, b, midpoint, #]&, steps]];
    Print["simpson sum:  ",
          Map[integral[func, a, b, simpson, #]&, steps]];
    Print["NIntegrate:  ", NIntegrate[func[x], {x, a, b}]] ]
```

Then we run it with an integrand and an interval as follows.

```
    In[6]:= compare[Sin, 0,N[Pi/2]]
left sum: {0.790766, 0.89861, ..., 0.991796, 0.992125}
```

```
right sum: {1.18347, 1.09496, ..., 1.00816, 1.00783}
trapezoidal sum: {0.987116, 0.996785, ..., 0.999978, 0.999979}
midpoint sum: {1.00645, 1.00161, ..., 1.00001, 1.00001}
simpson sum: {1.00001, 1., ..., 1., 1.}
NIntegrate: 1.
```

It follows from both sections 3 and 4 that we develop programs according to the following process:

1. Make functions higher order by *Currying*, thereby more general.
2. Apply them with actual function arguments to each element of a list, distribute them over a list or use other higher order functions over any expressions to get results.

Higher order functions without explicit typing in *Mathematica* instead of function arguments with type structure in C make this process easier.

5 Other functional features in *Mathematica*

There are some problems arising from the relation between higher order functions and other functional features in *Mathematica*. We discuss one of them and resolve them partially using *Mathematica*. It is important to note that *Mathematica* provides *meta–programming*, the perspective that functions themselves can be analyzed and modified as objects.

Mathematica is a type–free language. Though it is very convenient for developing programs in the interpreted environment of *Mathematica*, we must keep in mind types of both functions and their arguments. *Patterns*, the restriction on arguments of a function, can be used as first–order types, such as the declaration of `integral`. Though *Mathematica* may provide patterns for higher order types, we do not use them. So, we have developed a function which indicates the type of its argument. It works as follows:

```
In[7]:= Type[Sin]
Out[7]= Func[Real, Real]
In[8]:= Type[Sin[0.0]]
Out[8]= Real
```

Underlying theory for the function `Type` is the following simple type theory based on typed λ–calculus.

- Each atomic expression `a` has a predefined type A.
- If an atomic expression `x` has type A and an expression `M` type B, then `Function[x, M]` has type $A \to B$.

- If expressions `M` and `N` have types $A \to B$ and A, respectively, then `M[N]` has type B.

Currently, `Type` requires type information on intrinsic functions because we have not yet worked out how to extract type information from them. Presently, `Type` can not recognize builtin attributes for functions, other patterns including conditions and it works only on *DownValues* and *OwnValues*. Moreover, there exists another kind of restriction which is due to the undecidability of types for derived expressions, e.g., flow control, rules and scopes. Though `Type` can analyze only a few mathematical functions because of above restrictions, the main part of it is of the form:

```
SetAttributes[findtype, {HoldFirst}]
findtype[func_[args__]] :=
  Module[{...},
    funtype = findtype[func];
    argtypes = Map[findtype, {args}];
    If[matchtype[domain[funtype], argtypes],
       range[funtype],
       ...]; ...]
findtype[Function[sym_Symbol, body_]] :=
  Module[{...},
    {bodytype, symtype} = findtypewithsym[body, x];
    If[!uncertan[bodytype] && !uncertain[symtype],
       Func[symtype, bodytype],
       ...]; ...]
```

Higher order functions help us to develop programs analyzing functions for this problem and other problems raised from, e.g., the evaluation strategy.

6 Conclusion

To conclude this paper, we give some results of student questionnaires. Some of my questionnaires are related to functions arguments for those who learned only C or C++ (Group 1), to higher order functions for those who learned only *Mathematica* (Group 2) and to their comparison for those who learned C++ and then used *Mathematica* in my exercises as T.As (Group 3).

Though it was difficult for most students in Group 1 to handle function arguments in only the half term, some of them who acquired this facility added comments such as, “They encouraged us to make functions such as numerical integrations introduced in Section 3”.

Some students in the second group answered, “Though we have only few knowledges about higher order functions, we have often used them”, “As there are many higher order functions over lists, it is difficult to understand

their behaviors and handle them entirely". And those who had already learnt C added comments, "It is easer and simpler to translate codes from C into *Mathematica*, such as `For` loop for `for` loop, rather than to rewrite them using `Map` and `Apply`", "Though higher order functions on lists takes some time to master them, it is well worth the effort".

Students in the third group answered, "If we learned *Mathematica*, we could go forward", "Higher order programming with *Mathematica* is easier but more powerful than that with C++".

Though some students who could already handle C have taken time to get used to higher order functions in *Mathematica*, in particular those over lists, most students have accomplished them. We therefore conclude that *Mathematica* is suitable for those students not only as an computer algebra system but also as a higher order programming language.

References

[1] H. P. Barendregt. *The Lambda Calculus, syntax and semantics, 2nd revised ed.* North Holland, Amsterdam, 1984.

[2] S. MacLane. *Categories for the working mathematician.* Springer–Verlag, New York/Berlin, 1972.

[3] P.-L. Curien. *Categorical combinators, sequential algorithms and functional programming.* Pitman, London, 1985.

[4] W.-C. Yang. Encompassing current mathematical software and technology in teaching and research. *Asian Technology Conference in Mathematics*, 1997.

[5] R. E. Maeder. *The Mathematica Programmer, II.* Academic Press, 1996.