Deconstructing Factorial:
Using Progressive Variations to Illustrate Language Features and Programming Techniques

Fritz Ruehr

Willamette University, 900 State Street, Salem, Oregon 97301
fruehr@willamette.edu

1 Introduction

It is common in Computer Science education to present students with a number of different concrete algorithms which implement the same abstract function, for example to highlight issues of efficiency in a Data Structures course. Sorting algorithms and dictionary look-up structures are among the traditional hobbyhorses in this context.

In this paper we present a series of implementations of a single, simple function (factorial) for a rather different purpose, namely to explore the breadth of stylistic variation possible in the Haskell language, from the simplest beginners' techniques to sophisticated methods which appear in recent research literature. On the beginners' end of the spectrum we employ such basic language features and programming techniques as patterns, higher-order functions, recursion combinators, accumulating parameters and continuations. Later examples make use of more advanced features and techniques, including functorial types, comonads, static computation and nested types.

Our approach is founded on two basic principles: first, we use a simple and familiar example throughout the presentation. Second, we follow a progression which exposes more and more of the structure of the underlying concepts of the basic algorithm (natural numbers, lists, recursion, etc.). In this way we hope to demonstrate the pedagogical value of variations on a simple theme.

See the *Acknowledgements* section below for details of a fuller version of this work [Rue].

2 Basic recursion, folds and scans

Computer Science students everywhere are introduced to the concept of recursion via the factorial function, typically followed by the Fibonacci function and, with luck, by Ackermann's function. Factorial is usually cast in a relatively positive light, e.g. on the basis of tail recursion, whereas Fibonacci's and Ackermann's functions are often trotted out as examples of the dangers of recursion. The typical mathematical definition of factorial is something like this,

$$\text{fac}(n) = \begin{cases} 1\,, & \text{if } n = 0 \\ n \cdot \text{fac}(n-1)\,, & \text{otherwise} \end{cases}$$

presuming the natural numbers as a domain (our earlier Haskell versions of factorial are well-defined only for non-negative, integer arguments; later versions will circumvent this issue).

Our initial definition of factorial in Haskell in Figure 1 makes use of conditional expressions, equality, a little arithmetic and, of course, recursion: in this form it appears much the same as it would in other languages. But in Haskell we might also choose to make the definition look more like the mathematical one, through the use of guarded clauses, as in the second example. (NB: we

group multiple versions of the factorial function, all under the name "`fac`", to conserve space: in practice, these definitions would need to be made in separate files or else suitably renamed.)

```
fac n = if (n==0) then 1
                  else n * fac (n-1)

fac n | (n==0)    = 1
      | otherwise = n * fac (n-1)

fac 0 = 1
fac n = n * fac (n-1)

fac  0    =  1
fac (n+1) = (n+1) * fac n
```

Fig. 1. Simple versions using conditionals, guards and pattern matching

We can bring out some basic intuitions about the inductive structure of the natural numbers using Haskell's pattern-matching features. The third of the definitions in Figure 1 depends on the ordering of the clauses, a bit of subtlety which is important when introducing patterns to students; the fourth definition is a bit more robust, but employs the much-ridiculed "n+k patterns".

Unfortunately, these basic versions of factorial convey little of the typical Haskell style, which depends crucially on data structures such as lists, and on higher-order functions, often to capture common patterns of recursion. These ideas are exemplified by the first definition in Figure 2, which uses a *right fold,* and which also serves to introduce such Haskell conveniences as *operator sections* and the *ellipsis notation.* The second definition essentially defines a left fold in terms of a right fold, in a rather subtle use of higher-order functions. The third definition demonstrates how a list of partially applied functions may be generated and composed, whereas the fourth (generously provided by Ross Paterson [Pat]) reveals a pleasing fold/unfold symmetry (we will expand on this idea below). Finally, the fifth definition shows how scans can be used to "memoize" a function, although the efficiency gains achieved in this case are not as dramatic as one might hope.

```
fac n = foldr (*) 1 [1..n]

fac n = foldr (\x g n -> g (x*n)) id [1..n] 1

fac n = foldr (.) id (map (*) [1..n]) 1

fac = foldr (*) 1 . unfoldr (\n-> guard (n>0) >> return (n,n-1))

facs = scanl (*) 1 [1..]
fac n = facs !! n
```

Fig. 2. Variations based on folds, unfolds and scans

3 Iteration, accumulation and continuations

Of course, the functional style does not exist in a vacuum: many students will come to it from a background of imperative programming. It may therefore be instructive to show how Haskell can simulate imperative style—not so much to recommend this approach, but perhaps to illustrate the relative power, flexibility and conciseness of Haskell. In the first variation in Figure 3, we simulate imperative code using a state-transition style, and going to some effort to mimic a Pascal or C syntax. The second version is similar in spirit, but employs a different initial state (and thus different transition function and completion check); it also uses a more concise, even cryptic, style.

Simulating state at this level is not typical of functional style: we can also express factorial using the techniques of *accumulating parameters* and *continuations*. These techniques are often used to improve the efficiency of functional code and to force evaluation order. Here their effect on efficiency is negligible, but they serve to introduce novel concepts in a simple and familiar context; their definitions are also similar enough to invite fruitful comparison.

```
fac n = result (until done next init)
        where init = (0,1)
              next   (i,m) = (i+1, m * (i+1))
              done   (i,_) = i==n
              result (_,m) = m

fac n = snd (until ((>n).fst) (\(i,m) -> (i+1,i*m)) (1,1))

facAcc a 0 = a
facAcc a n = facAcc (n * a) (n-1)
fac = facAcc 1

facCps k 0 = k 1
facCps k n = facCps (k.(n *)) (n-1)
fac = facCps id
```

Fig. 3. Variations based on iteration, accumulation and continuations

```
s f g x = f x (g x)
k x y = x
b f g x = f (g x)
c f g x = f x g
y f = f (y f)
cond p f g x = if p x then f x else g x

fac  = y (b (cond ((==) 0) (k 1)) (b (s (*)) (c b pred)))
```

Fig. 4. Fast combinatory version (but "cheating" on Y)

Interestingly, the most efficient of all the variations on factorial presented here, at least in terms

of Hugs' reduction counts, is one based on purely functional techniques, namely the combinatory version given in Figure 4. This efficiency is presumably due to a good match with Hugs' combinator-based implementation. (Ross Paterson notes [Pat] that the definition of `Y` used here is a bit of a cheat, but that it can be "fixed" using an appropriate nested type; see [Rue].)

4 Recursive and functorial datatypes

We can get more serious about recursion over natural numbers by defining them explicitly as a Haskell datatype, rather than depending on the built-in `Integer` type. The definitions in Figure 5 include the datatype and recursion combinators for both iteration and primitive recursion (the latter supporting a somewhat simpler definition of factorial).

```
data Nat = Zero | Succ Nat

iter z s  Zero    = z
iter z s (Succ n) = s (iter z s n)
primrec z s  Zero    = z
primrec z s (Succ n) = s n (primrec z s n)

plus n = iter n     Succ
mult n = iter Zero (plus n)

fac  = snd . iter (one, one) (\(a,b) -> (Succ a, mult a b))
fac' = primrec one (mult . Succ)
```

Fig. 5. Peano-style natural numbers, iteration and primitive recursion

```
newtype Mu f = Mu (f (Mu f))
um (Mu x) = x

cata f = f  . fmap (cata f) . um
ana  g = Mu . fmap (ana  g) . g

data N b = Zero | Succ b
nelim z s  Zero    = z
nelim z s (Succ n) = s n
instance Functor N where fmap f = nelim Zero (Succ . f)

type Nat = Mu N
zero = Mu   Zero
succ = Mu . Succ

plus n = cata (nelim n     succ   )
mult n = cata (nelim zero (plus n))
```

Fig. 6: A datatype for natural numbers in functorial style

Following the lead of the Squiggol/BMF community (see e.g., [Mal]), we can go further, defining both natural numbers and lists in terms of a unified scheme of type recursion. This "functorial" style, inspired by category theory, provides generic recursion combinators for all types defined in terms of the type recursor `Mu`. Figure 6 provides the basic type definitions and notions of *cata-* and *anamorphism,* while Figure 7 adds definitions for a functorial version of lists. Using these, we can define factorial in terms of functions for product and list-generation—`cata` and `ana` here correspond to generalized versions of the fold and unfold functions from above (note that our use of "`succ`" conflicts with that of the Haskell Prelude, requiring some renaming or hiding).

```
data L a b = Nil | Cons a b
lelim n c  Nil       = n
lelim n c (Cons a b) = c a b

instance Functor (L a) where
  fmap f = lelim Nil (\a b -> Cons a (f b))

type List a = Mu (L a)

prod = cata (lelim (succ zero) mult)
upto = ana  (nelim Nil (diag (Cons.succ)) . um)
diag f x = f x x

fac = prod . upto
```

Fig. 7. Functorial lists with cata- and anamorphisms

Our final variation in the functorial style is the very general definition in terms of *comonads, generalized catamorphisms* and so-called *zygomorphisms* as defined by Uustalu, Vene and Pardo in their paper on comonadic recursion [UVP]. Our version in Figure 8 follows theirs closely—see the original paper for details (note that we require the functorial definitions from Figure 6 above).

5 Type-based computation

The next couple of examples break from the general trend, which until now has focussed on drawing out the recursive structure of either the algorithm itself or the datatypes used in its expression. Instead, the following examples use type-level structure to effect computation or structural constraints at the value level, thus computing factorial somewhat indirectly. These examples illustrate some more sophisticated aspects of Haskell's type system and in fact stress its limits, making essential use of constructor classes, functional dependencies, nested datatypes and rank–2 polymorphism. (Note that we omit some required type declarations for reasons of space.)

The first type-level variation is based on Thomas Hallgren's use of *functional dependencies* [Hall]: it uses datatypes to represent the Peano constructors and *constructor classes* to represent functions over the resulting type-level natural numbers. Qualified type instances for the constructor classes amount to Prolog-like Horn clauses, so that type inference for appropriate terms will compute a factorial result via type unification: in Hugs for example, try the type inference request "`:t fac three`" (larger "numbers" than in the example can be defined similarly if desired).

```
data Prod e c = Pair c e
outl (Pair x y) = x
outr (Pair x y) = y
fork f g x = Pair (f x) (g x)

instance Functor (Prod e) where
  fmap g = fork (g . outl) outr

class Functor n => Comonad n where
  extr :: n a -> a
  dupl :: n a -> n (n a)

instance Comonad (Prod e) where
  extr = outl
  dupl = fork id outr

gcata dist f = extr . cata (fmap f . dist . fmap dupl)
zygo g = gcata (fork (fmap outl) (g . fmap outr))
fac = zygo Mu h where
      h  Zero             = succ zero
      h (Succ (Pair f n)) = mult f (succ n)
```

Fig. 8. Uustalu, Vene and Pardo's comonadic version of factorial

```
data Zero
data Succ n

type One   = Succ Zero
type Two   = Succ One
(zero,one,two,three) =  (zero,one,two,three)
                     :: (Zero,One,Two,Three)

class Add a b c | a b -> c where add :: a -> b -> c
instance              Add  Zero    b  b
instance Add a b c => Add (Succ a) b (Succ c)

class Mul a b c | a b -> c where mul :: a -> b -> c
instance                            Mul  Zero    b Zero
instance (Mul a b c, Add b c d) => Mul (Succ a) b d

class Fac a b   | a -> b   where fac :: a -> b
instance                                Fac  Zero    One
instance (Fac n k, Mul (Succ n) k m) => Fac (Succ n) m
```

Fig. 9. A version of factorial exploiting type-level, static arithmetic

Our final variation is, as far as I know, an original contribution, although it is inspired by ideas about the relationships between number and structure explored by Chris Okasaki [Oka] and Ralf Hinze [Hin]. The basic idea is to use a *nested datatype* to express the factorial structure: values of the defined type will have a shape roughly comparable to a list of trees, but where successive trees are required to contain a number of components equal to the factorial of their index in the list.

We begin by defining type constructors and value-level functions which mimic the Peano constructors, thus allowing us to simulate n-sequences for any fixed n. We then define a nested structure, called `Facs` below, which takes two type parameters: the first is used to pass forward a "numeric" type structure, but wrapped by a successor in each new recursive call. The second parameter is used as the running "list component" type, but is wrapped by a use of the first type parameter in each successive call, thus nesting an n-component structure within an (n+1)-sequence. (Note the pleasant symmetry between the definitions of the data structures and the values here.)

Applying `Facs` to the `Zero` type and a suitable component type yields a type whose values are (infinite) "lists" with the desired structure. A function can then be used to count the components of a `Facs` structure, thus implementing factorial indirectly via a value of the `Facs` type (note that the `count` and `sizes` functions require explicit type declarations omitted here).

```
data Zero   a = Nil
zero        a = Nil
data Succ n a = Cons a (n a)
succ      n a = Cons a (n a)

data Facs n a = Fac a (Facs (Succ n) (Succ n a))
facs :: (forall a. a -> n a) -> a -> Facs n a
facs      n a = Fac a (facs (succ n) (succ n a))

instance Functor Zero where
  fmap f Nil = Nil
instance Functor n => Functor (Succ n) where
  fmap f (Cons x y) = Cons (f x) (fmap f y)

class Folding n where
  fold :: (a -> b -> b) -> b -> n a -> b
instance Folding Zero where
  fold c n Nil = n
instance Folding n => Folding (Succ n) where
  fold c n (Cons x y) = c x (fold c n y)

next f = fold (+) 0 . fmap f
count f (Fac a m) = f a : count (next f) m
sizes = count (const 1)

fac n = sizes (facs zero ()) !! n
```

Fig. 10. A factorial-structured nested datatype

6 Conclusions

We have explored some progressive variations on the definition of the factorial function, showing that they can be used to illustrate a number of Haskell language features, programming techniques and styles. The companion web document [Rue] contains details, elaborations and further examples not provided here, including an approach based on a simple language interpreter and a contribution based on Church numerals.

Some of the techniques illustrated here may seem trivial or transparent in the context of the factorial example … but that is the point! The simplicity and familiarity of factorial and the momentum of the progressive development support the understanding of ideas which could be daunting when presented in terms of more elaborate examples or when standing alone. I have successfully applied the same approach to other pedagogical content, including presentations of quicksort and a longer example involving parsing, interpretation and compilation of a simple expression language.

Acknowledgements

As mentioned above, the ideas presented in this paper are derived from a light-hearted treatment of the same topic available on the world-wide web [Rue]. The original version was inspired by a presentation of Uustalu, Vene and Pardo's paper *Recursion Schemes from Comonads* [UVP] made by Iavor Diatchki at OGI PacSoft's Tuesday seminar series, and by Iavor's citation of an old standby of programming humor, *The Evolution of a Programmer* [EP]. I am indebted to Messrs. Uustalu, Vene and Pardo for their formidable comonadic version of factorial, to Ross Paterson for contributions and comments, to Iavor as an (unwitting) agent of synchronicity, and to numerous people for their research on Haskell language features, programming techniques and style.

References

[EP] *The Evolution of a Programmer,* traditional (author unknown). One version available at `http://www.cse.ogi.edu/~diatchki/jokes/programmer.html`.

[Hall] *Fun with Functional Dependencies,* Thomas Hallgren. *Joint Winter Meeting of the Departments of Science and Computer Engineering,* Chalmers University of Technology and Göteborg University, Varberg, Sweden, 2001.

[Hin] *Manufacturing Datatypes,* Ralf Hinze. *Journal of Functional Programming,* 11(5):493-524, September 2001.

[Mal] *Homomorphisms and Promotability,* Grant Malcom. In *Mathematics of Program Construction,* Springer-Verlag, June 1989.

[Oka] *Purely Functional Data Structures,* Chris Okasaki. Cambridge University Press, 1999.

[Pat] Personal communications, Ross Paterson, August 2001.

[Rue] *The Evolution of a Haskell Programmer,* Fritz Ruehr, July 2001-present. Available at `http://www.willamette.edu/~fruehr/haskell/evolution.html`.

[UVP] *Recursion Schemes from Comonads,* T. Uustalu, V. Vene and A. Pardo. *Nordic Journal of Computing,* 8(3):366-390, 2001.