Folding left and right over Peano numbers*

Olivier Danvy
Yale-NUS College & School of Computing
National University of Singapore
danvy@acm.org

May 2019

Abstract

This functional pearl illustrates the analogue of fold-left and fold-right for Peano numbers, i.e., natural numbers in base 1, and shows that unlike for lists, these two functionals are equivalent. For Peano numbers, replacing fold-right by fold-left and inlining fold-left therefore makes it straightforward to calculate tail-recursive functions with an accumulator out of non-tail recursive functions that were obtained via tupling or that use Kleene's insight. In combination with Ohori and Sasano's lightweight fusion, this equivalence provides assistance for inter-deriving non-tail-recursive functions and tail recursive functions with an accumulator, generically. The difference between primitive recursion and primitive iteration – namely to have or not to have access to the value to which the induction hypothesis applies – prevents the generalization of this equivalence from primitive iteration to primitive recursion.

Contents

---

*Extended version of a functional pearl to appear in the Journal of Functional Programming 29, e6, 2019. https://doi.org/10.1017/S0956796819000042

1 Folding left and right over lists

The functionals `fold_left_list` and `fold_right_list` are virtually as old as functional programming since Strachey discovered them in the early 1960's [5,14]. Today these two functionals provide a convenient toolset to verify whether a list-processing function is structurally recursive—if it can be expressed with either of these two functionals, it is structurally recursive—and if so how to abstract and instantiate its base case and its induction step (Section 1.1), in the presence or in the absence of an accumulator (Section 1.2).

1.1 Abstraction and instantiation

Prototypically, `fold_left_list` and `fold_right_list` abstract the primitive-iterative programming pattern over lists, in the presence and in the absence of an accumulator:

```
Definition fold_left_list (V W : Type) (n : W) (c : V -> W -> W) (vs : list V) : W :=
  let fix loop vs a :=
    match vs with
    | nil      => a
    | v :: vs' => loop vs' (c v a)
    end
  in loop vs n.

Definition fold_right_list (V W : Type) (n : W) (c : V -> W -> W) (vs : list V) : W :=
  let fix visit vs :=
    match vs with
    | nil      => n
    | v :: vs' => c v (visit vs')
    end
  in visit vs.
```

These two definitions are expressed in Gallina, the total functional programming language of the Coq proof assistant [1]. (The entirety of this pearl is formalized in Coq.)

For example, consider the standard powerset function that maps the representation of a set as the list of its elements (in any order and without repetition) to the representation of its powerset:

```
Definition powerset (V : Type) (vs : list V) : list (list V) :=
  let fix outer vs :=
    match vs with
    | nil      => nil :: nil
    | v :: vs' => let powerset_vs' := outer vs'
                  in let fix inner wss :=
                       match wss with
                       | nil        => powerset_vs'
                       | ws :: wss' => let c := inner wss'
                                       in (v :: ws) :: c
                       end
                     in inner powerset_vs'
    end
  in outer vs.
```

This powerset function is listless [15] in that all the lists it constructs are part of the result (in other words, it creates no intermediate, transitory lists). It is also structurally recursive and thus can be expressed using two instances of `fold_right_list`, yielding du Feu's definition [8]:

```
Definition powerset_right (V : Type) (vs : list V) : list (list V) :=
  fold_right_list V
                  (list (list V))
                  (nil :: nil)
                  (fun v powerset_vs' => fold_right_list (list V)
                                                         (list (list V))
                                                         powerset_vs'
                                                         (fun ws c => (v :: ws) :: c)
                                                         powerset_vs')
                  vs.
```

Conversely, inlining `fold_right_list` in this definition yields the standard version of `powerset` above.

1.2 Inequivalence in general

As first pointed out by Strachey [14], `fold_left_list` and `fold_right_list` are not equivalent in general, witness, e.g., Bird and Wadler's duality theorems [2]. That said, the order in the given lists may not matter, in which case either functional can be used. For example, if a given list represents a set and the order of elements in this list does not matter, one can replace each call to `fold_right_list` by a call to `fold_left_list` in du Feu's definition of the powerset function. Inlining `fold_left_list` yields the following definition where the inner instance of `fold_left_list` is defined locally to the outer one:

```
Definition powerset_left_inlined (V : Type) (vs : list V) : list (list V) :=
  let fix outer vs outer_a :=
    match vs with
    | nil      => outer_a
    | v :: vs' => outer vs' (let fix inner wss inner_a :=
                               match wss with
                               | nil        => inner_a
                               | ws :: wss' => inner wss' ((v :: ws) :: inner_a)
                               end
                             in inner outer_a outer_a)
    end
  in outer vs (nil :: nil).
```

Since `inner` is tail-recursive, we can relocate the recursive call to `outer` to its base case, using lightweight fusion [13]. The result is a tail-recursive version with an accumulator that one might be hard pressed to write by hand in the first place:

```
Definition powerset_left_inlined_and_fused (V : Type) (vs : list V) : list (list V) :=
  let fix outer vs outer_a :=
      match vs with
      | nil      => outer_a
      | v :: vs' => let fix inner wss inner_a :=
                          match wss with
                          | nil        => outer vs' inner_a
                          | ws :: wss' => inner wss' ((v :: ws) :: inner_a)
                          end
                    in inner outer_a outer_a
      end
  in outer vs (nil :: nil).
```

2 Folding left and right over Peano numbers

The functionals fold_left_nat and fold_right_nat are the analogues of fold_left_list and fold_right_list over Peano numbers, i.e., natural numbers in base 1. They too provide a convenient toolset to verify whether a numerical function is structurally recursive—if it can be expressed with either of these two functionals, it is structurally recursive—and if so how to abstract and instantiate its base case and its induction step (Section 2.1), in the presence or in the absence of an accumulator (Section 2.2).

2.1 Abstraction and instantiation

Each of fold_left_nat and fold_right_nat abstract the primitive-iterative programming pattern over Peano numbers, in the presence and in the absence of an accumulator:

```
Definition fold_left_nat (V : Type) (z : V) (s : V -> V) (n : nat) : V :=
  let fix loop n a :=
    match n with
    | O    => a
    | S n' => loop n' (s a)
    end
  in loop n z.

Definition fold_right_nat (V : Type) (z : V) (s : V -> V) (n : nat) : V :=
  let fix visit n :=
    match n with
    | O    => z
    | S n' => s (visit n')
    end
  in visit n.
```

These two definitions are particularly simple to write in Gallina where natural numbers are represented as Peano numbers, i.e., are either O or the successor S of a natural number.

For example, the addition of two natural numbers can be defined tail recursively with an accumulator or non-tail recursively with no accumulator:

```
Definition add_acc (n m : nat) : nat :=
  let fix loop n a :=
    match n with
    | O    => a
    | S n' => loop n' (S a)
    end
  in loop n m.

Definition add (n m : nat) : nat :=
  let fix visit n :=
    match n with
    | O    => m
    | S n' => S (visit n')
    end
  in visit n.
```

The first definition is an instance of fold_left_nat and the second of fold_right_nat:

```
Definition add_left (n m : nat) : nat :=
  fold_left_nat nat m S n.

Definition add_right (n m : nat) : nat :=
  fold_right_nat nat m S n.
```

Inlining fold_left_nat and fold_right_nat in the definitions of add_left and add_right yields the definitions of add_acc and add.

For another example, the evenness of a natural number can be defined tail recursively with an accumulator or non-tail recursively with no accumulator:

```
Definition evenp_acc (n : nat) : bool :=
  let fix loop n a :=
    match n with
    | O    => a
    | S n' => loop n' (neg a)
    end
  in loop n true.

Definition evenp (n : nat) : bool :=
  let fix visit n :=
    match n with
    | O    => true
    | S n' => neg (visit n')
    end
  in visit n.

Theorem soundness_of_evenp :
  forall n : nat, evenp n = true -> exists m : nat, m = 2 * n.

Theorem completeness_of_evenp :
  forall n : nat, evenp (2 * n) = true.
```

The first definition is an instance of `fold_left_nat` and the second of `fold_right_nat`:

```
Definition evenp_left (n : nat) : bool :=
  fold_left_nat bool true neg n.

Definition evenp_right (n : nat) : bool :=
  fold_right_nat bool true neg n.
```

Inlining `fold_left_nat` and `fold_right_nat` in the definitions of `evenp_left` and `evenp_right` yields the definitions of `evenp_acc` and `evenp`.

2.2 Equivalence in general

As it happens, `fold_left_nat` and `fold_right_nat` are equivalent in general:

```
Theorem equivalence_of_left_fold_nat_and_right_fold_nat :
  forall (V : Type) (z : V) (s : V -> V) (n : nat),
    fold_left_nat V z s n = fold_right_nat V z s n.
```

Therefore `add_left` and `add_right` are formally equivalent, and as a corollary, `add_acc` and `add` are formally equivalent too. By the same token, `evenp_left` and `evenp_right` are formally equivalent, and as a corollary, `evenp_acc` and `evenp` are formally equivalent too.

This equivalence theorem hinges on either of the following master lemmas:

```
Lemma about_fold_left_nat :
  forall (V : Type) (z : V) (s : V -> V) (n : nat),
    fold_left_nat V (s z) s n = s (fold_left_nat V z s n).

Lemma about_fold_right_nat :
  forall (V : Type) (z : V) (s : V -> V) (n : nat),
    fold_right_nat V (s z) s n = s (fold_right_nat V z s n).
```

N.B.: The analogue of `about_fold_right_nat` holds for lists but not the list analogue of `about_fold_left_nat`.

The intuition here is that applying `fold_left_nat z s` or `fold_right_nat z s` to a number n yields the same result as applying the n-fold composition of `s` to `z`:

$$\underbrace{\mathtt{s}(\mathtt{s}(\ldots\mathtt{s}(}_{n}\mathtt{z})\ldots))$$

In this light, `fold_left_nat` accumulates the result of applying `s` iteratively whereas `fold_right_nat` applies `s` recursively. In both cases, `s` is applied n times.

Since `fold_right_nat` is the functional counterpart of Church numerals [4], the theorem sheds some light on the oft-mentioned [7] equivalence of the two definitions of the Church successor function, $\lambda n.\lambda z.\lambda s.s\ (n\ z\ s)$ and $\lambda n.\lambda z.\lambda s.n\ (s\ z)\ s$. (Whether z comes before or after s in a Church numeral, like whether `z` comes before or after `s` in the definition of `fold_left_nat` and `fold_right_nat`, is a matter of taste. In practice, one tends to use the same order as in the definition of the corresponding data type. And in an induction proof, one tends to consider the base case(s) before the inductive step(s).)

2.3 Application to computing a power of 2

Revisiting the powerset example to compute its cardinality, we can morph the definition of `powerset_right` in Section 1 from lists to natural numbers to compute powers of 2:

```
Definition exp2_right (n : nat) : nat :=
  fold_right_nat nat 1 (fun exp2_i => fold_right_nat nat exp2_i S exp2_i) n.
```

The third argument of `fold_right_nat` adds its argument to itself, i.e., it multiplies its argument by 2:

```
Definition times2 := fun n => fold_right_nat nat n S n.

Theorem soundness_and_completeness_of_times2 : forall n : nat, times2 n = 2 * n.
```

The intuition here is that applying `exp2_right` to a number n yields the same result as applying the n-fold composition of `times2` to `1`:

$$\underbrace{\mathtt{times2}(\mathtt{times2}(\ldots\mathtt{times2}(}_{n}\mathtt{1})\ldots))$$

The equivalence theorem says that this computation can be achieved tail recursively by using `fold_left_nat` instead of `fold_right_nat`.

Inlining `fold_left_nat` yields the morphed counterpart of `powerset_left_inlined` in Section 1.2:

```
Definition exp2_left_inlined (n : nat) : nat :=
  let fix outer n outer_a :=
    match n with
    | O    => outer_a
    | S n' => outer n' (let fix inner m inner_a :=
                          match m with
                          | O    => inner_a
                          | S m' => inner m' (S inner_a)
                          end
                        in inner outer_a outer_a)
    end
  in outer n 1.
```

Since `inner` is tail-recursive, we can relocate the recursive call to `outer` to its base case, using lightweight fusion:

```
Definition exp2_left_inlined_and_fused (n : nat) : nat :=
  let fix outer n outer_a :=
    match n with
    | O    => outer_a
    | S n' => let fix inner m inner_a :=
                match m with
                | O    => outer n' inner_a
                | S m' => inner m' (S inner_a)
                end
              in inner outer_a outer_a
    end
  in outer n 1.
```

2.4 Application to tupled functions: the Fibonacci function

As initiated by Burstall and Darlington [3], one can obtain a linear-time function computing Fibonacci numbers by first defining a function that computes two consecutive such numbers:

```
Definition fibfib (n : nat) : nat * nat :=
  let fix visit n :=
    match n with
    | O    => (0, 1)
    | S n' => let (fib_n', fib_Sn') := visit n' in (fib_Sn', fib_n' + fib_Sn')
    end
  in visit n.

Definition fib (n : nat) : nat :=
  let (fib_n, fib_Sn) := fibfib n in fib_n.
```

Since `fibfib` fits the fold-right pattern, it can be expressed as such:

```
Definition fibfib_right (n : nat) : nat * nat :=
  fold_right_nat (nat * nat)
                 (0, 1)
                 (fun c => let (fib_i, fib_Si) := c in (fib_Si, fib_i + fib_Si))
                 n.

Definition fib_right (n : nat) : nat :=
  let (fib_n, fib_Sn) := fibfib_right n in fib_n.
```

Since `fold_left_nat` and `fold_right_nat` are equivalent, one can replace the other:

```
Definition fibfib_left (n : nat) : nat * nat :=
  fold_left_nat (nat * nat)
                (0, 1)
                (fun c => let (fib_i, fib_Si) := c in (fib_Si, fib_i + fib_Si))
                n.

Definition fib_left (n : nat) : nat :=
  let (fib_n, fib_Sn) := fibfib_left n in fib_n.
```

Inlining `fold_left_nat`, lambda-lifting `loop` [10], and inlining `fibfib_left` yields the familiar iterative definition of the Fibonacci function with a pair of accumulators:

```
Fixpoint fibfib_left_loop (n : nat) (a : nat * nat) : nat * nat :=
  match n with
  | O    => a
  | S n' => let (fib_i, fib_Si) := a in fibfib_left_loop n' (fib_Si, fib_i + fib_Si)
  end.
```

```
Definition fib_left_inlined_and_lifted (n : nat) : nat :=
  let (fib_n, fib_Sn) := fibfib_left_loop n (0, 1) in fib_n.
```

As usual, `fibfib_left_loop` begs to be inlined and its initial call to be lightweight-fused to become a tail call. Currying then yields the familiar iterative definition of the Fibonacci function with two accumulators:

```
Definition fib_left_inlined_and_fused_and_curried (n : nat) : nat :=
  let fix loop n fib_i fib_Si :=
    match n with
    | O    => fib_i
    | S n' => loop n' fib_Si (fib_i + fib_Si)
    end
in loop n 0 1.
```

2.5 Application to computing the prefix of a stream

Given a natural number representing a length and a stream, one constructs the prefix of this stream with this length by recursively traversing it at call time and constructing the resulting list at return time:

```
CoInductive stream (V : Type) : Type :=
| Cons : V -> stream V -> stream V.

Definition stream_prefix_right (V : Type) (n : nat) (vs : stream V) : list V :=
  fold_right_nat (stream V -> list V)
                 (fun vs => nil)
                 (fun c vs => match vs with
                              | Cons _ v vs' => v :: c vs'
                              end)
                 n
                 vs.
```

The equivalence theorem says that this computation can be achieved tail recursively by accumulating a "stream prefixer" and eventually applying it to the given stream.

2.6 A first application of Kleene's insight: the predecessor function

Let us revisit Church numerals. When the world was young [4], it was unclear how to lambda-define the predecessor function until Kleene, while at the dentist [11], anticipated the tupling strategy by computing a pair of Church numbers, one of them the predecessor of the other, when it exists. His insight makes it possible, given a positive number n, to apply $n-1$ times a given `s` over a given `z`, without breaking the abstraction. Here is the `fold_right_nat` analogue:

```
Definition predecessor_right (n : nat) : option nat :=
  let (on', S_n') := fold_right_nat (option nat * nat)
                                    (None, 0)
                                    (fun p => let (_, S_i) := p
                                              in (Some S_i, S S_i))
                                    n
  in on'.

Theorem soundness_and_completeness_of_predecessor_right :
  forall n n' : nat,
    predecessor_right n = Some n' <-> n = S n'.
```

The equivalence theorem says that this simulation can be achieved tail recursively with a pair of accumulators.

N.B.: In actuality, Kleene's explanation [11] is not based on pairs, but on triples, which is an artifact of using the λ-I calculus. Our exposition here is aligned with the modern rendition of the story put forward in Goldberg's lecture notes on the λ calculus [7].

2.7 A second application of Kleene's insight: the first suffix of a list

Just as `fold_right_nat` is the functional counterpart of Church numerals, `fold_right_list` is the functional counterpart of the Church encoding of lists [4]. And as it turns out, Kleene's insight can be applied, e.g., to computing the first suffix (i.e., the tail) of a list, if this list is non-empty, by constructing a pair of lists, one of them the tail of the other when it exists:

```
Definition list_suffix_right (V : Type) (vs : list V) : option (list V) :=
  let (ovs', vs) := fold_right_list V
                                    (option (list V) * list V)
                                    (None, nil)
                                    (fun v c => let (_, vs') := c
                                                in (Some vs', v :: vs'))
                                    vs
  in ovs'.

Theorem soundness_of_list_suffix_right :
  forall (V : Type) (v : V) (vs : list V) (ows : option (list V)),
    list_suffix_right V (v :: vs) = ows -> ows = Some vs.

Theorem completeness_of_list_suffix_right :
  forall (V : Type) (vs : list V) (v : V) (vs' : list V),
    vs = v :: vs' -> list_suffix_right V vs = Some vs'.
```

However, due to the inequivalence between `fold_right_list` and `fold_left_list`, replacing one by the other yields the first prefix of the given list in reverse order if this list is non-empty.

2.8 A third application of Kleene's insight: the first prefix of a list

Kleene's insight also translates to computing the first prefix of a list, if this list is non-empty, by constructing a pair of difference lists [9], one of them the optional prefix of the other, and eventually applying the second one to the empty list to construct the prefix:

```
Definition list_prefix_left (V : Type) (vs : list V) : option (list V) :=
  let (odl', dl) := fold_left_list V
                                   ((list V -> option (list V)) * (list V -> list V))
                                   ((fun vs => None), (fun vs => vs))
                                   (fun v c => let (_, dl') := c
                                               in ((fun vs => Some vs) `o` dl',
                                                   dl' `o` (cons v)))
                                   vs
  in odl' nil.
```

where `` `o` `` is an infix notation for function composition.

Again, due to the inequivalence between `fold_left_list` and `fold_right_list`, replacing one by the other yields the first suffix of the given list in reverse order if this list is non-empty.

2.9 A fourth application of Kleene's insight: the factorial function

Let us get back to folding over Peano numbers. In contrast to Gödel's System T, the abstracted induction step does not have access to the current snapshot of the given number in the course of the calls (which is where primitive recursion and primitive iteration differ, as outlined in Appendix A). This lack of access makes it non-obvious to express, e.g., the factorial function since it uses the successive decrements of the given number to construct its result. In his PhD thesis, Goldberg [6] used Kleene's insight to lambda-define the factorial function over Church numerals: he returned a pair containing an index i and the factorial of i. Here is the `fold_right_nat` analogue:

```
Definition specification_of_the_factorial_function (fac : nat -> nat) : Prop :=
  fac 0 = 1 /\ forall n' : nat, fac (S n') = S n' * fac n'.
```

```
Definition factorial_right (n : nat) : nat :=
  let (n, fac_n) := fold_right_nat (nat * nat)
                                   (0, 1)
                                   (fun c => let (i, fac_i) := c in (S i, S i * fac_i))
                                   n
  in fac_n.

Theorem factorial_right_satisfies_the_specification_of_the_factorial_function :
  specification_of_the_factorial_function factorial_right.
```

The equivalence theorem says that this simulation can be achieved tail recursively with a pair of accumulators, making it clear that Goldberg crystallized Kleene's insight as enumerating the graph of the given function using Church numerals.

2.10 More applications of Kleene's insight: the atoi function, etc.

Likewise, the atoi function that maps a number to a list of its successive predecessors can be simulated by enumerating its graph:

```
Definition atoi_right (n : nat) : list nat :=
  let (n, ns) := fold_right_nat (nat * list nat)
                                (0, nil)
                                (fun c => let (i, is) := c in (S i, i :: is))
                                n
  in ns.
```

The equivalence theorem says that this simulation can be achieved tail recursively with a pair of accumulators.

In the same spirit, it is simple to list the successive suffixes of a list and the successive difference lists of its prefixes (or directly of its successive prefixes), though there, as in Sections 2.7 and 2.8, this listing is order-sensitive.

3 Left or right?

Why would one prefer recursion over tail recursion with an accumulator, or vice versa? For one point, fold-right-based functions tend to be simpler to reason about and also more amenable to deforestation than fold-left-based ones. On the other hand, tail-recursive functions are famed to be more efficient to implement.

For example, consider the case of the `fibfib` function in Section 2.4:

- Here is the invariant for its recursive instance:

  ```
  Lemma about_fibfib_right :
    forall n : nat,
      fibfib_right n = (reference_fib n, reference_fib (S n)).
  ```

  where `reference_fib` is a reference definition of the Fibonacci function.

- And here is the invariant for its tail-recursive counterpart:

  ```
  Lemma about_fibfib_left_loop :
    forall i j : nat,
      fibfib_left_loop i (reference_fib j, reference_fib (S j)) =
      (reference_fib (i + j), reference_fib (S (i + j))).
  ```

As one can see, the first invariant is simpler than the second, which illustrates the relevance of theorems about folding left and right.

Acknowledgments: The author is grateful to Andreea Costea, Kira Kutscher, and Ilya Sergey for their encouragement. Thanks are also due to the anonymous reviewers for insightful comments and to Ralf Hinze and Jeremy Gibbons for their editorship.

References

[1] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development.* Springer, 2004.

[2] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice-Hall International, London, UK, 1st edition, 1988.

[3] Rod M. Burstall and John Darlington. A transformational system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[4] Alonzo Church. *The Calculi of Lambda-Conversion.* Princeton University Press, 1941.

[5] Olivier Danvy and Michael Spivey. On Barron and Strachey's Cartesian product function, possibly the world's first functional pearl. In Ralf Hinze and Norman Ramsey, editors, *Proceedings of the 2007 ACM SIGPLAN International Conference on Functional Programming (ICFP'07)*, SIGPLAN Notices, Vol. 42, No. 9, pages 41–46, Freiburg, Germany, September 2007. ACM Press.

[6] Mayer Goldberg. *Recursive Application Survival in the λ-Calculus.* PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, May 1996.

[7] Mayer Goldberg. The λ calculus – outline of lectures. `http://lambda.little-lisper.org`, August 2014.

[8] Michael J. C. Gordon. On the power of list iteration. *The Computer Journal*, 22(4):376–379, 1979.

[9] John Hughes. A novel representation of lists and its application to the function "reverse". *Information Processing Letters*, 22(3):141–144, 1986.

[10] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 190–203, Nancy, France, September 1985. Springer-Verlag.

[11] Stephen C. Kleene. Origins of recursive function theory. *Annals of the History of Computing*, 3(1):52–67, January 1981.

[12] Lambert Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

[13] Atsushi Ohori and Isao Sasano. Lightweight fusion by fixed point promotion. In Matthias Felleisen, editor, *Proceedings of the Thirty-Fourth Annual ACM Symposium on Principles of Programming Languages*, SIGPLAN Notices, Vol. 42, No. 1, pages 143–154, Nice, France, January 2007. ACM Press.

[14] Christopher Strachey. Handwritten notes. Archive of working papers and correspondence. Bodleian Library, Oxford, Catalogue no. MS. Eng. misc. b.267., 1961.

[15] Philip Wadler. Listlessness is better than laziness. In Guy L. Steele Jr., editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 282–305, Austin, Texas, August 1984. ACM Press.

A Primitive iteration and primitive recursion

As pointed out in Section 2.9, fold-right functionals embody primitive iteration because the abstracted induction step does not have access to the current snapshot of the given value in the course of the calls. In contrast, a 'parafold-right' functional (i.e., a paramorphism [12]) embodies primitive recursion by enabling the abstracted induction step to have access to this current snapshot:

```
Definition parafold_right_nat (V : Type) (z : V) (s : nat -> V -> V) (n : nat) : V :=
  let fix visit n :=
    match n with
    | O    => z
    | S n' => s n' (visit n')
    end
  in visit n.
```

For example, applying `parafold_right_nat z s` to `3` yields `s 2 (s 1 (s 0 z))`, where `s` is applied 3 times. It is thus simple to define the atoi function from Section 2.10, i.e., the function that maps a natural number to the decreasing list of its predecessors, without using Kleene's insight:

```
Definition atoi_right_alt (n : nat) : list nat :=
  parafold_right_nat (list nat) nil (fun i c => i :: c) n.
```

This primitive-recursive functional also makes it simple to define, e.g., the predecessor function and the factorial function without using Kleene's insight. It is not equivalent to its parafold-left counterpart for the same reason that `fold_right_list` is not equivalent to `fold_left_list`, even though either can be used to define the other:

```
Definition parafold_left_nat (V : Type) (z : V) (s : nat -> V -> V) (n : nat) : V :=
  let fix loop n a :=
    match n with
    | O    => a
    | S n' => loop n' (s n' a)
    end
  in loop n z.
```

For example, applying `parafold_left_nat z s` to `3` yields `s 0 (s 1 (s 2 z))`, where `s` is applied 3 times. It is thus simple to define a iota function, i.e., a function that maps a natural number to the increasing list of its predecessors:

```
Definition iota_left (n : nat) : list nat :=
  parafold_left_nat (list nat) nil (fun i c => i :: c) n.
```

("atoi" is "iota" spelled backwards. These names come from APL.)

Programmatically, the word "iteration" in "primitive iteration" might be misleading since "iteration" does not mean "tail recursion" here. For example, the same point applies, e.g., to binary trees—namely to have or not to have access to the value to which the induction hypothesis applies—and binary trees are not traversed tail-recursively when they are processed by a function which is structurally recursive, be it primitive iterative or primitive recursive. Binary trees have no fold-left functionals.

B Primitive recursion in Coq

In Gallina, the type `nat` pre-exists and was defined as an inductive data type. Therefore `parafold_right_nat` also pre-exists, under the name of `nat_rect`:

```
Definition parafold_right_nat_rect (V : Type) (z : V) (s : nat -> V -> V) (n : nat) : V :=
  nat_rect (fun (_ : nat) => V) z s n.
```

As pointed out in Appendix A, `parafold_right_nat`, and therefore `nat_rect`, make it simple to define, e.g., the predecessor function and the factorial function without using Kleene's insight. Their soundness is proved by induction:

```
Definition predecessor_rect (n : nat) : option nat :=
  nat_rect (fun (_ : nat) => option nat) None (fun n' _ => Some n') n.

Theorem soundness_of_predecessor_rect :
  forall n n' : nat,
    predecessor_rect n = Some n' -> n = S n'.

Definition factorial_rect (n : nat) : nat :=
  nat_rect (fun (_ : nat) => nat) 1 (fun i fac_i => S i * fac_i) n.

Theorem factorial_rect_satisfies_the_specification_of_the_factorial_function :
  specification_of_the_factorial_function factorial_rect.
```

We notice, though, that in the `Init/Nat.v` library, `iter`, the primitive iterator for natural numbers, is defined with `nat_rect`. This definition seems like an overkill since `nat_rect` embodies primitive recursion, not primitive iteration. The main result of this pearl suggests instead to reason with `fold_right_nat` for simplicity and to compute with `fold_left_nat` for efficiency when iterating over Peano numbers.