On Eliminating Nat

@December 25, 2021

👉 This article can be treated as an extended version of "The Little Typer"

Nat stands for Natural Number.

In Pie, four eliminators of Nat: `which-Nat`, `iter-Nat`, `rec-Nat` and `ind-Nat` are introduced for safe (guarded) recursive style. And we know in ordinary typed languages like Coq, pattern matching is chosen as a primitive and only way to eliminate data constructor. To demonstrate two different flavours of eliminating, I am going to desugar these four Nat eliminators in a pattern matching style in Coq, and see how it captures the idea of safe recursion which ensures that function always terminates.

💡 Termination Checking: Dependent Types at Work (Chapter 2.7)

Let's start with which-Nat, introduced as the first eliminator for Nat. The usage of which-Nat is simply removing its constructor (add1 in Pie, S in Coq) and then applying operation on a smaller one. It's primitive in Pie and can be defined in Coq below:

```
Definition which_nat {A : Type} (target : nat) (base : A) (step : nat -> A) : A :=
match target with
  | O   => base
  | S n => step n
end.
```

Believe it or not, it's all we need for implementing any opeartions on Nat if you are familiar with recursion. For example addition of two Nats:

```
Fixpoint addition (n : nat) (m : nat) : nat :=
  which_nat n m (fun p => S (addition p m)).
```

We can compare it with the official implementation in Coq.

```
Nat.add =
fix add (n m : nat) : nat := match n with
                             | 0   => m
                             | S p => S (add p m)
end.
```

Obviously authors are not satified with single which-Nat because "recursion is not an option" as they repeated. Unrestricted recursion could lead program never terminate. Coq owns positivity check to avoid such situations but that's not appropriate for a minimial language like Pie. Thus they introduce iter-Nat, which captures a guarded recursion pattern.

```
Fixpoint iter_nat {A : Type} (target : nat) (base : A) (step : A -> A) : A :=
match target with
  | O   => base
  | S n => step (iter_nat n base step)
end.
```

Thus the addition could be implemented, returning the ability of recursion back to language itself.

```
Definition addition_iter (n : nat) (m : nat) : nat :=
  iter_nat n m S.
```

We can go back to check the implementation of Nat.add again: the abstracted part become smaller: only S (add1 in Pie). It can be understood that which-Nat destructs n and then pass the smaller n to the lambda function (3rd argument); iter-Nat destructs n and pass the result of orginal funtion with same set of arguments (but with smaller n) to the lambda function.

Let's give it a try to apply the idea of iter-Nat to the implementation of gauss addition, we shall know: iter-Nat desturcts Nat and pass the result of functional call with smaller Nat to the function.

```
Definition gauss_iter (n : nat) : nat :=
  iter_nat n 0 (fun r => n + r).
```

Dumbs like me could've somehow implemented gauss_iter above but it's completely wrong. Check the correct pattern matching implementation:

```
Fixpoint gauss_pm (n : nat) : nat :=
match n with
  | O    => 0
  | S n1 => S n1 + gauss_pm n1
end.
```

Oops, it seems that we pass a constant n to inter_nat instead of one that keeps changing in recursion. Like statement in the The Little Typer (Page 77), "gauss needs an eliminator that combines the expressiveness of both which-Nat and iter-Nat". Then here comes rec-Nat.

```
Fixpoint rec_nat {A : Type} (target : nat) (base : A) (step : nat -> A -> A) : A :=
match target with
  | O   => base
  | S n => step n (rec_nat n base step)
end.
```

gauss appears naturally:

```
Definition gauss_rec (n : nat) : nat :=
  rec_nat n 0 (fun n' r => S n' + r).
```

ind-Nat, the last one, is in a differnt story that about dependent elimination.

And the classical example of dependent types is List type with length indexed, we name it Vect here.

```
(1 :: Nil)           : (Vect 1 Integer)
(1 :: 1 :: Nil)      : (Vect 2 Integer)
(1 :: 1 :: 1 :: Nil) : (Vect 3 Integer)
```

Let's try to define a function ones that generates a list of integer 1 with specified number n.

```
ones : (n : Nat) -> Vect n Integer
ones n = iter_nat n Nil (1 ::)
```

Well, the problem lies in the above code is mismatch of types:

```
base   : A
step   : A -> A
Nil    : Vect 0 Integer
(1 ::) : Vect n Integer -> Vect (S n) Integer
```

So we introduce ind-Nat, which is rec-Nat for dependent types.

```
Fixpoint ind_nat (target : nat) (mot : nat -> Type) (base : mot O)
  (step : forall (n : nat), mot n -> mot (S n)) : mot target :=
match target with
  | O      => base
  | (S n') => step n' (ind_nat n' mot base step)
end.
```

Compared with rec_nat, there're some type-level computations on base, step and result type repectively.

```
Fixpoint iter_nat {A : Type} (target : nat) (base : A) (step : A -> A) : A :=
match target with
  | O   => base
  | S n => step (iter_nat n base step)
end.
```

Additional Readings

```
forall (n : Nat), mot n -> mot (S n)
```

should have no differences with (except default implicitness in Coq)

```
(n : Nat) -> mot n -> mot (S n)
```

Dependent types allows us to define more expressive types, we can view some types we deal with everyday as special cases of it. For example,

```
map : (a -> b) -> [a] -> [b]
```

is a special case of

```
map : (a : Type) -> (b : Type) -> (a -> b) -> [a] -> [b]
```

we can also change it into implicit parameters form in order to make it feel less verbose:

```
map : {a : Type} -> {b : Type} -> (a -> b) -> [a] -> [b]
```

Godel System T

```
natrec : {C : Set} -> C -> (Nat -> C -> C) -> Nat -> C
```