structure table of contents and add warning
mhe authored 1 week ago

8e420126

LazyNaturals.md 3.75 KB

Lazy natural numbers

We introduce the lazy natural numbers, with a sample application to make a certain algorithm faster.

Motivating example

If the list `xs` is large, the following is slow. Moreover, it loops without giving an answer if the list `xs` is infinite:

```
checkLengthBiggerThan :: [a] -> Int -> Bool
checkLengthBiggerThan xs n = length xs > n
```

Examples:

```
*Main> :set +s
*Main> checkLengthBiggerThan [1..10^6] 3
True
(0.04 secs, 72,067,432 bytes)
*Main> checkLengthBiggerThan [1..10^7] 3
True
(0.19 secs, 720,067,488 bytes)
*Main> checkLengthBiggerThan [1..10^8] 3
True
(1.47 secs, 7,200,067,600 bytes)
*Main> checkLengthBiggerThan [1..10^9] 3
True
(14.35 secs, 72,000,067,640 bytes)
```

We can make this faster as follows, in such a way that it will take at most `n` steps, regardless of the length of `xs`.

```
checkLengthBiggerThan' :: [a] -> Int -> Bool
checkLengthBiggerThan' []     0 = False
checkLengthBiggerThan' xs     0 = True
checkLengthBiggerThan' []     n = False
checkLengthBiggerThan' (x:xs) n = checkLengthBiggerThan' xs (n-1)
```

We are ignoring negative numbers.

Example:

```
*Main> checkLengthBiggerThan' [1..10^9] 3
True
(0.01 secs, 68,408 bytes)
```

Lazy natural numbers

There is another way to make the above fast, by replacing the use of type `Int` by the type `Nat` of lazy natural numbers in the original algorithm:

```
data Nat = Zero | Succ Nat deriving (Eq,Ord)
```

The idea is that this represents natural numbers 0,1,2,3,... as follows

```
        Zero
        Succ Zero
        Succ (Succ Zero)
        Succ (Succ (Succ Zero))
        ...
```

which we can define in Haskell as

```
one, two, three :: Nat
one   = Succ Zero
two   = Succ one
three = Succ two
```

Moreover, we can convert a non-negative integer to a lazy natural number as follows:

```
toNat :: Int -> Nat
toNat 0 = Zero
toNat n = Succ (toNat (n-1))
```

But we also have infinity in the type `Nat`:

```
infty = Succ infty
```

This computes for ever, producing and infinite pile `Succ (Succ (Succ (Succ ...` of successor constructors, but, crucially, this computation is lazy.

We can now define a length function as follows:

```
length' :: [a] -> Nat
length' []     = Zero
length' (x:xs) = Succ (length' xs)
```

For example, we have that

- `length [0..]` loops without giving any answer, but
- `length' [0..] = infty`.

Now define a lazy comparison algorithm as follows:

```
biggerThan :: Nat -> Nat -> Bool
Zero     `biggerThan` y        = False
(Succ x) `biggerThan` Zero     = True
(Succ x) `biggerThan` (Succ y) = x `biggerThan` y
```

The point is that in the second equation, `x` doesn't need to be evaluated in order to give the answer `True`. For example:

```
*Main> infty `biggerThan` three
True
```

With this, the first algorithm becomes fast, and also works for infinite lists:

```
checkLengthBiggerThan'' :: [a] -> Int -> Bool
checkLengthBiggerThan'' xs n = (length' xs) `biggerThan` (toNat n)
```

For example:

```
*Main> checkLengthBiggerThan'' [1..10^9] 3
True
(0.02 secs, 69,032 bytes)
```

But actually we can write this as follows, because we derived `Ord`:

```
checkLengthBiggerThan''' :: [a] -> Int -> Bool
checkLengthBiggerThan''' xs n = length' xs > toNat n
```

And this is just as fast, which means that the derivation mechanism for `Ord` must produce a lazy comparison algorithm similar to ours:

```
*Main> checkLengthBiggerThan''' [1..10^9] 3
True
(0.01 secs, 70,200 bytes)
```

The point is that the use of the lazy natural numbers makes the natural algorithm fast, which is slow for integers. Moreover, an advantage of the use of the lazy natural numbers is that it makes the length of an infinite list well defined, because of the availability of the lazy natural number `infty`