Natural Number Arithmetic Examples Using Peano-Inspired Structures

H. Conrad Cunningham

11 April 2022

Contents

Professor of Computer and Information Science
University of Mississippi
214 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-7396 (dept. office)

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of April 2022 is a recent version of Firefox from Mozilla.

Natural Number Arithmetic Examples

Background

In this set of examples, we develop representations for the natural numbers and their key operations using type and data structures inspired by Peano's Postulates [3,4], a well-know axiomization of the natural numbers. None of the representations can use any builtin integer type.

Mathematically, we can define the set `Nat` (of natural numbers) inductively as follows:

```
x in Nat if and only if
     (1) x = 0                                  -- zero
  or (2) (Exists y : y in Nat :: x = Succ(y)) -- successor
```

Note: We consider 0 as being a natural number as is customary for many computer scientists.

The `Nat` values use a unary representation for the natural numbers; there is one `successor` object in a linear structure for each natural number value greater than 0.

I have developed these examples in various ways for various languages, but in all cases we seek to use functional techniques in the following ways:

- Natural number instances can be created and examined but not modified; the storage for instances no longer accessible can be reclaimed (i.e., garbage collected).
- The operations are pure functions; they can return new natural number instances or other values, but they have have no side effects on their operands or other aspects of the program state.

In the examples, I tend to use one of the following program organizations:

- modules of pure functions that operate on algebraic data types or data structures that operate similarly
- traditional object-oriented class hierarchies structured according to the Composite, Singleton, and Null Object software design patterns [1,2,5].

Haskell

This problem is used in exercises in Chapters 21 (Algebraic Data Types) and 25 (Haskell Laws) of the book *Exploring Languages with Interpreters and Functional Programming*. I first devised this exercise in the early/mid 1990s to ask students to implement simple algebraic data types in my Functional Programming course.

Scala

This set includes three Scala 2 implementations of the Natural Numbers with the following characteristics:

- Functional object-oriented with ordinary classes — Scala source

  This 2008+ Scala version defines a traditional object-oriented class hierarchy organized according to the Composite, Singleton, and Null Object software design patterns. Once created, the `Nat` objects are immutable. The operations do not modify the state of an object; they create new objects with the modified state. The program carries out computations by "passing messages" among the objects, taking advantage of subtype polymorphism (dynamic binding) to associate method calls with the correct implementation. This version also encapsulates the state within the objects in the hierarchy.

- Functional module with case classes – Scala source

  This 2012+ Scala version uses an algebraic data type (defined using a Scala sealed trait implemented by a group of case classes/objects) with a set of pure functions defined in the companion object for the trait. This version uses structural pattern matching to identify the correct operation functionality. By using cases classes/objects, this version exposes the state outside the hierarchy.

  Note: The `Nat` algebraic data type is organized similarly to the `List` algebraic data type in Chapter 3 (Functional Data Structures) of the *Notes on Functional Programming in Scala*.

- Functional object-oriented with case classes — Scala source

  This 2012+ Scala version is in between the implementations above. Like the first, it uses a traditional object-oriented structure but, like the second, it uses case classes/objects instead of ordinary classes/objects to take advantage of some of the features of case classes/objects. It uses subtype polymorphism for the left argument and uses pattern matching for the right argument to select the correct operation functionality. By using cases classes/objects, this version exposes the state outside the hierarchy.

TODO: Update the old Scala 2 programs to be compatible with Scala 3.

Elixir

This set includes one Elixir implementation from 2015. It structures the program as a module of pure functions. It represents the natural numbers as Elixir tuples with constants (i.e., Elixir atoms) in the first component and uses structural pattern matching to deconstruct the tuples based on the constant values. (This simulates an algebraic data type mechanism.)

The implementation also seeks to facilitate the possible use of different data representations by separating the module into primitive (private) aspects that can manipulate the data representation directly and nonprimitive (public) that carry out the arithmetic operations using the primitive aspects. These layers should be broken into two separate modules.

- Nat module

  test module

  test script

TODO: Update the 2015 Elixir code to be compatible with the current release of Elixir.

Lua

The set includes one Lua implementation from 2013/14. It structures the program as a module of pure functions. It represents the natural numbers as Lua tables with constants in the first component and uses structural pattern matching to deconstruct them. It thus simulates an algebraic data type as Lua tables with tags in the first position. It implements the operations in a functional style. I also stresses modularity, with primitive and nonprimitive layers as discussed above.

- source `nats2.lua`

TODO: Update the Lua 5.1 program from 2014 to be compatible with the current Lua 5.4 release.

Ruby

The set includes one Ruby implementation from 2006. It defines a traditional object-oriented class hierarchy organized according to the Composite, Singleton, and Null Object software design pattern. The hierarchy has a base class `Nat` and subclasses `Zero`, `Succ`, and `Err`.

- source `Nat.rb`

TODO: Update this old 2006 code for the current version of Ruby.

Java

The set includes one Java version, whose code was originally written in 2004 using a release of Java with no generics. It uses a traditional object-oriented structure as described above.

- abstract base class `Nat`
- subclass `Zero`

- subclass `Succ`
- subclass `Err`
- main program `TestNat`

Acknowledgements

I often use this problem when I am learning and/or teaching programming language. I have done that with (at least) Haskell, Java, Ruby, Scala, Lua, and Elixir since sometime in the 1990s. This chapter collects the source code for most of these efforts.‘

I retired from the full-time faculty in May 2019. As one of my post-retirement projects, I am continuing work on possible textbooks based on the course materials I had developed during my three decades as a faculty member. In January 2022, I began refining the existing content, integrating separately developed materials together, reformatting the documents, constructing a unified bibliography (e.g., using citeproc), and improving my build workflow and use of Pandoc. I adapted this index page from a portion of my Spring 2019 CSci 555 course notes.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, Boston, Massachusetts, USA.

[2] Source Making. 2022. Design patterns. Retrieved from https://sourcemaking.com/design_patterns

[3] Wikpedia: The Free Encyclopedia. 2022. Peano axioms. Retrieved from https://en.wikipedia.org/wiki/Peano_axioms

[4] Wolfram Research, Inc. 2022. Peano's axioms. Retrieved from https://mathworld.wolfram.com/PeanosAxioms.html

[5] Bobby Woolf. 1997. Null object. In *Pattern languages of program design 3*, Robert Martin, Dirk Riehle and Frank Buschmann (eds.). Addison-Wesley, Boston, Massachusetts, USA, 5–18.