Integers in Lua 5.3

Р. Иерусалимский
PUC-Rio

Pontifícia Universidade Católica do Rio de Janeiro

Numbers in Lua

- Since its first version (1993), Lua has had one single kind of number
- First versions used `float`
- Changed to double in version 3.1 (1998)
  - mainly because programmers needed 32-bit values
  - a float has only 24 bits of mantissa, a double has 53 bits.

Doubles

GOOD

- Well-defined rules (IEEE), including error and overflow handling (±inf, NaN)
- Hardware support in conventional platforms
  - even in 1998
- 53 bits is enough for most counting purposes
  - 1 petabyte
  - 1 million times the world population
  - 300 000 years in seconds
  - 20% of total global wealth in cents of dollars

Doubles

BAD

- Big and slow for restricted hardware
- Awkward for bitwise operators
  - should they operate on 53 bits?
  - ~0 is 0xFFFFFFFF or -1?
- Some algorithms need 64 bits
  - cryptography, encodings
- Some data need 64 bits
  - handles

Doubles

BAD

- Integers already present in Lua as second-class values.
  - several library functions use integers (e.g., indices in the string library)
  - conversions not well specified and/or not efficient
  - `string.sub(s, -3.4, 8.7)`
- Confusing in the C API
  - conversions always lose bits in some direction

Integers

- 64-bit values
- Several options:
  - `long double`
  - infinite precision (e.g., Python)
  - a new type (e.g., UInt64 in Javascript)
  - inside type `number`, not exposed to the programmer (e.g., LNUM in Lua)
  - as a subtye of `number`, exposed to the programmer

Long Double

- Offers 64 bits
- Keeps simplicity and elegance of IEEE
- Fully compatible
- Only small changes in the implementation

GOOD

Long Double

- More problematic for small machines
  - and even for not-so-small ones
- Increases memory use
- Not part of C89 standard
- Even C99 does not require a long double to be really “long”
- Not widely supported (e.g., MS VS...)

BAD

Integers: Infinite Precision

- Elegant
- Avoid problems with signed x unsigned
- Safe

GOOD

Integers: Infinite Precision

- Quite Expensive
- Not that useful in practice
  - when compared with 64 bits
- Problem in the C API

BAD

64-bit Data as a New Type

- Keeps the simplicity of IEEE arithmetic
- Few changes in the language
- Solves the problem of 64-bit data

GOOD

64-bit Data as a New Type

- Does not solve the other problems...
  - restricted hardware, 64-bit algorithms, bitwise operations, interfaces with integers

BAD

Integers as “Implementation Detail”

- Keeps an apparent simplicity
- Solves all problems in our list
- Allows *Lua-32*
  - uses 32-bit integers plus single floats

GOOD

Integers as “Implementation Detail”

- Somewhat expensive
- No explicit control for the programmer
- Complex rules for arithmetic operations
  - (2^62 + 2) * 0.5 = ?

    (All operands have exact representations, result has exact representation, but operation does not give the exact result.)

BAD

Integers as a Subtype

- Explicit difference between 1 and 1.0
- Almost transparent to programmers
  - automatic coercion between floats and integers
- “[The] programmer has the option of mostly ignore the difference between integers and floats or assume complete control about the representation of each value.”

Lua 5.3 reference manual

Main Rules

- Quite conventional
- Integer and float values are explicitly different things
  - `print(1, 1.0)  -->  1    1.0`
- Values of both subtypes have type `number`
  - `print(type(1), type(1.0))`

    `--> number  number`
- Coercion makes them quite similar
  - `print(1 == 1.0)  --> true`

Guidelines

- The subtype of the result of an operation can depend on the subtypes of its arguments, but it should not depend on the *values* of its arguments
    - easier for tools and for humans to infer subtypes
- Operations on reals under which integers are closed should be polymorphic:
    - `3.0 + 5.0 ≡ 8.0`
    - `3 + 5 ≡ 8`
    - `3.0 + 5 ≡ 8.0`   (real is the more general type)
    - similar for `-`, `*`, `%`

Other Operations: Division

- Avoid nightmare of `3/2 ≡ 1` but `3.0/2 ≡ 1.5`
- Two separated operations: float division (`/`) and integer division (`//`)
  - Like in Python
- Integer division converts *operands* to integers and does an integer division
  - mainly because it is simpler than otherwise
  - otherwise, what about `((2^62 + 2) // 2.0)`?

Other Operations: Exponentation

- What to do with negative integer exponents, such as `(3 ^ -2)`?
- `3^2` is integer but `3^-2` is float?
  - Violates guideline 1
- Pretend that `(3 ^ -2) ≡ (1 // 3^2)`?
  - complex and useless
- Operation is always on floats
  - integer exponentiation is useful, but not enough to deserve its own operator

Coercions

- Integers are always valid where floats are expected: convertion never fails
- Floats can be converted to integers when its value does not change (that is, it has an integral value in the proper range)

```
string.sub(s, 1.5)
   stdin:1: bad argument #2 to 'sub'
   (number has no integer representation)
```

Integer Overflows

- Different cases:
  - constants
  - conversion from floats
  - operations
- Different options:
  - convert to floats
  - error
  - wrap around

Overflow: Constants

- Convert to float: weird and useless
- Error:
  - a little tricky for unsigned integers
  - programs for 64-bit Lua may not even compile in Lua-32!
- Wrap around
  - dangerous
  - solves the problem for unsigned

Overflow: Conversion from Floats

- Error seems a good option here
  - not a common operation
  - other behaviors not useful

Overflow: Integer Operations

- Convert to float
  - not as useful as it seems
  - good for compatibility
  - expensive
- Errors
  - kills unsigned arithmetic
  - expensive
- Wrap around
  - allows unsigned arithmetic
  - cheap

Bitwise Operators

- Absence of integers was *the* reason for the absence of bitwise operators in Lua
- Mostly conventional: &, |, ~, >>, <<
- Operates on 64 bits
- a~b for exclusive or
  - a^b already taken
- >> is logical shift
  - no arithmetic shift; use arithmetic operation (integer division)

Other Aspects

- Numerals: decimal point or exponent makes a float; otherwise number is integer
  - `0.0    1e1    0xFFF.0`
  - `0    234    0xFFF`
- `print` distinguishes between floats and integers (!)
- Table keys: float keys with integer values are converted to integers
  - `a[1.0] = 0; print(next(a)) --> 1   0`

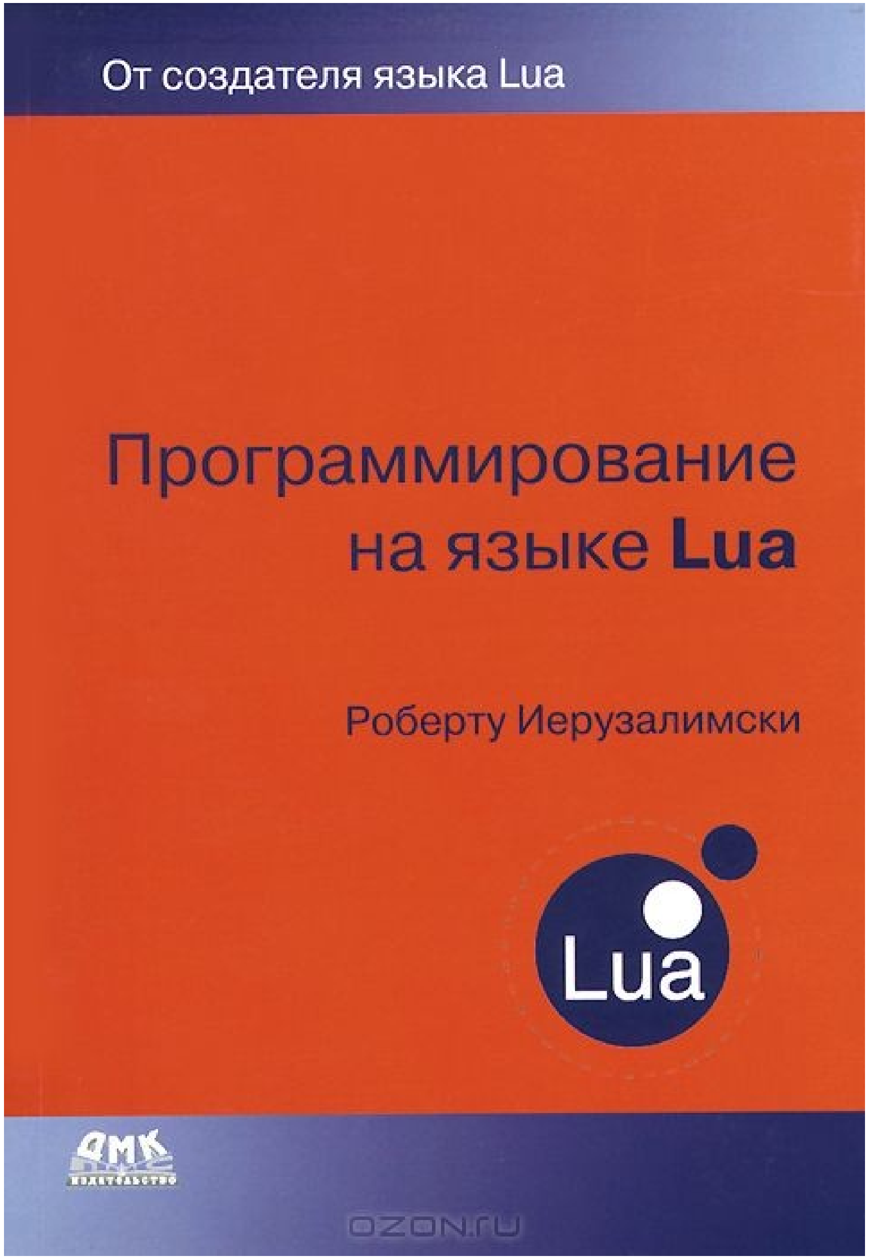
Other Aspects

- `tonumber` and `io.read("n")` return float or integer depending on the numeral’s syntax
  - `tonumber(“1”) --> 1`

    `tonumber(“1.0”) --> 1.0`
  - breaks guideline 1

Final Remarks

- People loved the bitwise operators :-)
- Mostly compatible with 5.2
  - main problem: `print(1.0) --> 1.0`
- Code base clearer and more conformant with ANSI C
  - coercions from floats to integers
- Seems to satisfy original goals
- Lua-32 will be officially supported



От создателя языка Lua

Программирование на языке Lua

Роберту Иерузалимски