bignums implementation

Alexandre Rademaker

June 18, 2025

Contents

1 predicate allZero

Predicate that checks if a string is empty or consists only of `0` characters.

```
predicate allZero(s: string)
```

1.1 auto-informalized

The allZero predicate is a fundamental validation function that determines whether a string represents the mathematical value zero in bit string form. It accepts a string parameter and returns true if the string is either empty or contains only '0' characters.

The predicate handles edge cases gracefully:

- Empty strings are considered to represent zero
- Strings containing only '0' characters (regardless of length) represent zero
- Mixed strings containing both '0' and '1' characters do not represent zero

2 function zeros

Creates a bit string of n zeros that evaluates to `0`.

```
method zeros(n: nat) returns (s: string)
 ensures |s| == n
 ensures balidBitString(s)
 ensures Str2Int(s) == 0
 ensures AllZero(s)
```

2.1 auto-informalized

The zeros function is a constructor that creates bit string representations of the mathematical value zero with a specified length. It takes a natural number parameter n and returns a string of exactly n '0' characters.

The function maintains correctness through formal postconditions:

- Ensures the result has exactly n characters in length
- Guarantees the result is a valid bit string (contains only '0' and '1' characters)
- Confirms the string evaluates to the integer value 0 when interpreted
- Verifies the result satisfies the allZero predicate

3 predicate validBitString

Predicate that checks if a string consists only of `0` and `1` characters.

```
predicate validBitString(s: string)
```

3.1 auto-informalized

The `validBitString` predicate is a fundamental validation function that determines whether a string represents a well-formed bit string suitable for big integer arithmetic operations. It accepts a string parameter and returns true if the string contains only valid binary digits ('0' and '1' characters) and is non-empty.

The validation criteria are:

- The string must not be empty (ensures meaningful representation)
- Every character must be either '0' or '1' (ensures valid binary representation)
- No other characters are permitted (rejects malformed input)

The predicate provides the foundation for safe big integer operations by ensuring all input strings represent valid binary numbers before any arithmetic processing begins.

4 function normalizeBitString

Removes leading zeros from a bit string (except keeps at least one digit), ensuring the result is a valid bit string with the same integer value (if the original bitstring was also valid).

```
method normalizeBitString(s: string) returns(t: string)
 ensures ValidBitString(t)
 ensures |t| > 0
 ensures |t| > 1 ==> t[0] != '0'
 ensures ValidBitString(s) ==> Str2Int(s) == Str2Int(t)
```

4.1 auto-informalized

The normalizeBitString function performs canonical normalization on bit string representations of big integers. It takes a string parameter that may contain leading zeros and returns a normalized version that represents the same integer value while conforming to standard bit string formatting.

The function maintains correctness through formal preconditions and postconditions:

- Postconditions guarantee the result is a valid bit string with at least one digit, no leading zeros (unless the string represents zero), and equivalent integer value to the input (when the input is valid)

The normalization process removes unnecessary leading zeros while preserving the mathematical value, ensuring consistent representation of integers. This is essential for:

- Canonical comparison operations between bit strings
- Efficient storage by eliminating redundant leading zeros
- Maintaining deterministic output format for arithmetic operations
- Ensuring bit string representations follow standard conventions

This specification enables verification of normalization operations in big integer libraries, making it suitable for cryptographic applications, arbitrary precision arithmetic systems, and other domains requiring consistent canonical representation of large numbers.

5 function add

If the input is valid, then the output is valid and the output string interpreted as an integer equals the sum of the input strings interpreted as integers.

```
method Add(s1: string, s2: string) returns (res: string)
 requires ValidBitString(s1) && ValidBitString(s2)
 ensures ValidBitString(res)
 ensures Str2Int(res) == Str2Int(s1) + Str2Int(s2)
```

5.1 auto-informalized

The `add` function performs arithmetic addition on two big integers represented as bit strings. It takes two string parameters (s1 and s2) that must be valid bit string representations of integers, and returns their sum as another valid bit string.

The function maintains correctness through formal preconditions and postconditions:

- Preconditions ensure both input strings are valid bit representations
- Postconditions guarantee the result is also a valid bit string and that when interpreted as an integer, it equals the mathematical sum of the input integers

This specification enables verification of big integer addition operations where the integers may exceed the capacity of standard integer types, making it suitable for cryptographic applications, arbitrary precision arithmetic, and other domains requiring large number computations.

6 function mul

If the input is valid, then the output is valid and the output string interpreted as an integer equals the product of the input strings interpreted as integers.

```
method Mul(s1: string, s2: string) returns (res: string)
 requires ValidBitString(s1) && ValidBitString(s2)
 ensures ValidBitString(res)
 ensures Str2Int(res) == Str2Int(s1) * Str2Int(s2)
```

6.1 auto-informalized

The Mul function performs arithmetic multiplication on two big integers represented as bit strings. It takes two string parameters (s1 and s2) that must be valid bit string representations of integers, and returns their product as another valid bit string.

The function maintains correctness through formal preconditions and postconditions:

- Preconditions ensure both input strings are valid bit representations
- Postconditions guarantee the result is also a valid bit string and that when interpreted as an integer, it equals the mathematical product of the input integers

This specification enables verification of big integer multiplication operations where the integers may exceed the capacity of standard integer types. The multiplication algorithm must handle arbitrary precision arithmetic efficiently, making it suitable for cryptographic applications, RSA operations, and other domains requiring large number computations where multiplication performance is critical.

7 function sub

If the inputs are valid and the first string is greater than or equal to the second (when interpreted as integers), then the output is valid and equals the difference between the first and second strings (as integers)

```
method Sub(s1: string, s2: string) returns (res: string)
 requires ValidBitString(s1) && ValidBitString(s2)
 requires Str2Int(s1) >= Str2Int(s2)
 ensures ValidBitString(res)
 ensures Str2Int(res) == Str2Int(s1) - Str2Int(s2)"
```

7.1 auto-informalized

The Sub function performs arithmetic subtraction on two big integers represented as bit strings. It takes two string parameters (s1 and s2) that must be valid bit string representations of integers, and returns their difference as another valid bit string.

The function maintains correctness through formal preconditions and postconditions:

- Preconditions ensure both input strings are valid bit representations and that s1   s2 (to avoid negative results)
- Postconditions guarantee the result is also a valid bit string and that when interpreted as an integer, it equals the mathematical difference of the input integers

This specification enables verification of big integer subtraction operations where the integers may exceed the capacity of standard integer types. The subtraction algorithm must handle arbitrary precision arithmetic efficiently while ensuring non-negative results, making it suitable for cryptographic applications, modular arithmetic, and other domains requiring large number computations where subtraction operations are constrained to non-negative results.

8 function div

Performs division on bit strings, returning both quotient and remainder as valid bit strings. The input divisor (as an integer) must be nonzero

```
method DivMod(dividend: string, divisor: string) returns (quotient: string, remainder
 requires ValidBitString(dividend) && ValidBitString(divisor)
 requires Str2Int(divisor) > 0
 ensures ValidBitString(quotient) && ValidBitString(remainder)
 ensures Str2Int(quotient) == Str2Int(dividend) / Str2Int(divisor)
 ensures Str2Int(remainder) == Str2Int(dividend) % Str2Int(divisor)
```

8.1 auto-informalized

The `div` function performs integer division on two big integers represented as bit strings, simultaneously computing both the quotient and remainder. It takes two string parameters (dividend and divisor) that must be valid bit string representations, and returns two results: the quotient and remainder as separate valid bit strings.

The function maintains correctness through formal preconditions and postconditions:

- Preconditions ensure both input strings are valid bit representations and that the divisor represents a positive integer (preventing division by zero)

- Postconditions guarantee both results are valid bit strings, with the quotient representing the integer division result and the remainder representing the modulo operation result

This specification enables verification of big integer division operations where the integers may exceed standard integer type capacities.

9 function ModExp

Computes modular exponentiation ($\mathrm{sx}^{\mathrm{sy}}$ mod sz) for bit strings. The inputs must be valid, and the output will be valid. The modulus must be at least 2.

```
method ModExp(sx: string, sy: string, sz: string) returns (res: string)
 requires ValidBitString(sx) && ValidBitString(sy) && ValidBitString(sz)
 ensures ValidBitString(res)
 ensures Str2Int(res) == Exp_int(Str2Int(sx), Str2Int(sy)) % Str2Int(sz)
 requires |sy| > 0 && Str2Int(sz) > 1
```

9.1 auto-informalized

The ModExp function performs modular exponentiation on three big integers represented as bit strings, computing $\mathrm{sx}^{\mathrm{sy}}$ mod sz. It takes three string parameters (sx, sy, sz) that must be valid bit string representations, and returns the result as another valid bit string.

The function maintains correctness through formal preconditions and postconditions:

- Preconditions ensure all three input strings are valid bit representations, that the exponent sy is non-empty, and that the modulus sz represents an integer greater than 1 (preventing division by zero or one in modular arithmetic)

- Postconditions guarantee the result is a valid bit string that represents the mathematical value of sx raised to the power sy, modulo sz

10 function compare

Compares two bit strings and returns -1, 0, or 1 if the first is less than, equal to, or greater than the second (as integers)

```
method compare(s1: string, s2: string) returns (res: int)
 requires ValidBitString(s1) && ValidBitString(s2)
 ensures Str2Int(s1) < Str2Int(s2) ==> res == -1
 ensures Str2Int(s1) == Str2Int(s2) ==> res == 0
 ensures Str2Int(s1) > Str2Int(s2) ==> res == 1
```

10.1 auto-informalized

The compare method performs ordered comparison between two big integers represented as bit strings. It takes two string parameters (s1 and s2) that must be valid bit string representations, and returns an integer indicating their relative ordering.

The method maintains correctness through formal preconditions and postconditions:

- Preconditions ensure both input strings are valid bit representations
- Postconditions establish a complete ordering relationship:
  - Returns -1 when s1 represents a smaller integer than s2
  - Returns 0 when s1 and s2 represent equal integer values
  - Returns 1 when s1 represents a larger integer than s2

This specification enables verification of comparison operations for big integers that may exceed standard integer type capacities. The comparison algorithm must handle arbitrary precision arithmetic efficiently while providing deterministic ordering results.

11 lemma IgnoreInitialZeros

Proves that leading zeros in a bit string don't affect its value.

```
lemma IgnoreInitialZeros(s : string, numZeros:int)
 requires ValidBitString(s)
 requires 0<=numZeros<=|s|
 requires forall i :: 0<=i<numZeros ==> s[i] == '0'
 ensures Str2Int(s) == Str2Int(s[numZeros..])
```

11.1 auto-informalized

The IgnoreInitialZeros lemma establishes that removing leading zeros from a valid bit string does not change its mathematical value.

The lemma takes two parameters: a string ‘s‘ and an integer ‘numZeros‘. It requires that ‘s‘ is a valid bit string, that ‘numZeros‘ is between 0 and the length of ‘s‘ (inclusive), and that all characters at positions 0 through ‘numZeros-1‘ are ’0’ characters.

Under these conditions, the lemma guarantees that the integer value of the original string ‘s‘ equals the integer value of the substring starting from position ‘numZeros‘ to the end (‘s[numZeros..]‘).

This fundamental property enables safe normalization of bit strings by proving that leading zeros can be removed without affecting the represented integer value, which is essential for canonical representation and efficient comparison operations in big integer arithmetic.

12 lemma TrailingZeros

Shows that trailing zeros in a bit string multiply its value by powers of 2.

```
lemma TrailingZeros(s: string, numZeros: nat)
 requires ValidBitString(s)
 requires numZeros <= |s|
 requires forall i :: |s| - numZeros <= i < |s| ==> s[i] == '0'
 ensures Str2Int(s) == Str2Int(s[..|s|-numZeros]) * Pow2(numZeros)
```

12.1 auto-informalized

The TrailingZeros lemma establishes that trailing zeros in a valid bit string correspond to multiplication by powers of 2.

The lemma takes two parameters: a string ‘s‘ and a natural number ‘numZeros‘. It requires that ‘s‘ is a valid bit string, that ‘numZeros‘ is at most the length of ‘s‘, and that all characters at the last ‘numZeros‘ positions (from position ‘|s| - numZeros‘ to ‘|s| - 1‘) are ’0’ characters.

Under these conditions, the lemma guarantees that the integer value of the original string ‘s‘ equals the integer value of the prefix substring (excluding the trailing zeros) multiplied by 2 raised to the power of ‘numZeros‘. Specifically, ‘Str2Int(s) == Str2Int(s[..|s|-numZeros]) * Pow2(numZeros)‘.

This fundamental property captures the mathematical effect of trailing zeros in binary representation: each trailing zero effectively shifts the bit pattern left by one position, which corresponds to multiplication by 2. Multiple trailing zeros result in multiplication by successive powers of 2, enabling efficient representation and manipulation of numbers that are multiples of powers of 2 in big integer arithmetic systems.

13 lemma Eleven

Proves that the bit string 1011 represents the decimal value 11.

```
lemma Eleven()
  ensures Str2Int("1011") == 11
```

13.1 auto-informalized

The Eleven lemma demonstrates the correctness of binary-to-decimal conversion for a specific example. It establishes that the bit string `1011` correctly evaluates to the decimal value 11.

This lemma serves as a concrete verification example showing how the Str2Int function performs binary interpretation. The bit string `1011` represents:

- Position 0 (rightmost): $1 \times 2 \;= 1 \times 1 = 1$
- Position 1: $1 \times 2^1 = 1 \times 2 = 2$
- Position 2: $0 \times 2^2 = 0 \times 4 = 0$
- Position 3 (leftmost): $1 \times 2^3 = 1 \times 8 = 8$

Summing these positional values: $8 + 0 + 2 + 1 = 11$.

14 comments

- `divMod` was renamed to `div` since the spec says nothing about the result being module something. The original name `divMod` means that it performs division and also modulo.
- In the specification and Dafny, an empty string is a `validBitString` but a `normalizedBitString` is a `validBitString` with no leading zeros and no empty.
- It is not clear why we need `CompareUnequal` since it is just a special case of `compare` that can handle strings with different length.