Computational Logic Optional Homework

Stefan Stefanache (916/2)

December 11, 2020

1 Statement

The task was to develop an application that implements algorithms for:

- Arithmetic operations for positive integers in a base:
  - ★ Addition
  - ★ Subtraction (with the subtrahend greater than the minuend)
  - ★ Multiplication
  - ★ Division
- Conversion of natural numbers between two bases $p, q \in \{2, 3, ..., 16\}$, using:
  - ★ The rapid conversion method for $p, q \in \{2, 4, 8, 16\}$
  - ★ The successive division method (used for $p < q$)
  - ★ The substitution method (used for $p > q$)

2 Solution

The provided solution consists of a C++ toolset that implements the features mentioned above, wrapped under a Qt5 user interface. Therefore, the given archive contains:

- Source code
- Catch2 test cases source code
- Windows and Linux executables x64
- qmake Makefile
- Doxygen generated documentation PDF
- This file

For a more in-depth view, check the source code or the documentation PDF.

3 Code Analysis

3.1 Number

The **Number** data-type encapsulates the concept of number representation in a base. The main attributes of the type are its base and its value string. (All number values are represented as strings). Besides basic features (constructors, getters), the interface provides the needed arithmetic operations.

Let's take a look at the interface provided in **src/number.hpp** (for further details and specifications look directly into the source file/documentation PDF):

```cpp
class Number {
    private:
        std::string value_string;
        static std::string get_removed_zero_padding(const std::string& value_string);
        static std::string get_zero_padded(const std::string& value_string,
                                           size_t target_length);
    public:
        const unsigned int base;

        // Constructors
        Number(const unsigned int base, const std::string& value_string);
        Number(const Number& other);
        Number(Number&& other);

        // Value handlers
        std::string get_value() const;
        static bool validate_value_string(const unsigned int base,
                                          const std::string& value_string);

        // Equality operator
        bool operator==(const Number& other) const;

        // Assignment operators
        Number& operator=(const Number &other);
        Number& operator=(Number &&other);

        // Arithmetic operators
        Number operator+(const Number &other) const;
        Number operator-(const Number &other) const;
        Number operator*(const Number &digit) const;
        std::pair<Number, Number> operator/(const Number &digit) const;
};
```

3.2 Arithmetic Operations

3.2.1 Addition

Pseudocode The addition algorithm is the one used in class, which has the following pseudocode:

Algorithm 1: Addition of two numbers in the same base

```
1  function addNumbers ($a, b, base$);
   Input  : the used base $base$,
            the numbers $a, b$
   Output: the number c resulted from the sum
2  make $a$ and $b$ have the same number of digits by adding 0s to the left of the shorter one
3  let $r \leftarrow 0$
4  for $i \leftarrow length(a) - 1$ to $0$: do
5      let $sum \leftarrow$ toDecimal($a[i]$) + toDecimal($b[i]$) + toDecimal($r$);
6      insertDigitAtBeginning($c$, fromDecimal($base, sum$ div $base$);
7      $r \leftarrow$ fromDecimal($base, sum$ mod $base$);
8  end
9  if $r \neq 0$ then
10     insertDigitAtBeginning($c$, fromDecimal)$r$);
11 end
```

Code Translated to our code base, we get the following implementation:

```cpp
Number Number::operator+(const Number& other) const {
    if (this->base != other.base)
        throw std::runtime_error("Operands must be of the same base");

    const std::string base_characters = get_base_characters(this->base);

    // Add '0' padding on the left such that both strings are the same length
    const std::string value =
        get_zero_padded(this->value_string, other.get_value().size());
    const std::string other_value =
        get_zero_padded(other.get_value(), this->value_string.size());

    std::string result_value = "";
    unsigned int remainder = 0;
    for (int char_index = value.size() - 1; char_index >= 0; char_index--) {
        unsigned int digit_sum_value = digitToValue(value[char_index]) +
                                       digitToValue(other_value[char_index]) +
                                       remainder;

        result_value.insert(result_value.begin(),
                            valueToDigit(digit_sum_value % this->base));

        remainder = digit_sum_value >= this->base ? 1 : 0;
    }

    if (remainder)
        result_value.insert(result_value.begin(), '1');

    return Number(this->base, result_value);
}
```

3.2.2 Subtraction

Pseudocode The subtraction algorithm has the following pseudocode:

Algorithm 2: Subtraction of two numbers in the same base (with subtrahend > minuend)

```
1  function subtractNumbers ($a, b, base$);
   Input  : the used base $base$,
            the numbers $a, b$ with $a > b$
   Output: the number c resulted from the subtraction
2  make $a$ and $b$ have the same number of digits by adding 0s to the left of the shorter one
3  let $r \leftarrow 0$
4  for $i \leftarrow 0$ to $length(a) - 1$: do
5      let $d \leftarrow$ toDecimal($a[i]$) - toDecimal($b[i]$) - toDecimal($r$);
6      if $d \geq 0$ then
7          insertDigitAtBeginning($c$, fromDecimal($base, d$);
8          $r \leftarrow 0$;
9      end
10     else
11         insertDigitAtBeginning($c$, fromDecimal($base, d$ + toDecimal($base$));
12         $r \leftarrow 1$;
13     end
14 end
```

Code Our code becomes:

```cpp
Number Number::operator-(const Number& other) const {
    if (this->base != other.base)
        throw std::runtime_error("Operands must be of the same base");
    if (other.get_value().size() > this->value_string.size())
        throw std::runtime_error("Difference can't be negative");

    const std::string base_characters = get_base_characters(this->base);
    const std::string value =
        get_zero_padded(this->value_string, other.get_value().size());
    const std::string other_value =
        get_zero_padded(other.get_value(), this->value_string.size());

    std::string result_value = "";
    unsigned int remainder = 0;
    for (int char_index = value.size() - 1; char_index >= 0; char_index--) {
        int digit_diff_value = digitToValue(value[char_index]) -
                               digitToValue(other_value[char_index]) -
                               remainder;
        if (digit_diff_value >= 0)
            result_value.insert(result_value.begin(), valueToDigit(digit_diff_value));
        else
            result_value.insert(result_value.begin(),
                                valueToDigit((base_characters.size() + digit_diff_value)));

        remainder = digit_diff_value < 0 ? 1 : 0;
    }

    if (remainder)
        throw std::runtime_error("Difference can't be negative");

    return Number(this->base, result_value);
}
```

3.2.3 Multiplication By Digit

Pseudocode The multiplication algorithm has the following pseudocode:

Algorithm 3: Multiplication of a number by a digit (same base)

```
1  function multiplyNumbers ($a, b, base$);
   Input  : the used base $base$,
            the numbers $a, b$ with $b$ being a digit
   Output: the number c resulted from the multiplication
2  let $r \leftarrow 0$
3  for $i \leftarrow length(a) - 1$ to $0$: do
4      let $p \leftarrow$ toDecimal($a[i]$) * toDecimal($b[i]$) + toDecimal($r$);
5      insertDigitAtBeginning($c$, fromDecimal($base, p$ div base)
6      $r \leftarrow p$ mod $base$;
7  end
8  if $r \neq 0$ then
9      insertDigitAtBeginning($c$, fromDecimal($base, r$)
10 end
```

Code Our code becomes:

```
Number Number::operator*(const Number& digit) const {
    if (this->base != digit.base)
        throw std::runtime_error("Operands must be of the same base");

    if (digit.get_value().size() != 1)
        throw std::runtime_error("Multiplication can only be done by DIGIT");

    std::string result_value = "";
    unsigned int remainder = 0;
    unsigned int multiplication_digit = digitToValue(digit.get_value()[0]);

    for (int char_index = this->value_string.size() - 1; char_index >= 0; char_index--) {
        unsigned int product = digitToValue(this->value_string[char_index]) * multiplication_digit
                               + remainder;

        remainder = product / this->base;

        result_value.insert(result_value.begin(), valueToDigit(product % this->base));
    }

    if (remainder)
        result_value.insert(result_value.begin(), valueToDigit(remainder));

    return Number(this->base, result_value);
}
```

3.2.4 Division By Digit

Pseudocode The division algorithm has the following pseudocode:

Algorithm 4: Division of a number by a digit (same base)

```
1 function divideNumbers ($a, b, base$);
  Input  : the used base $base$,
           the numbers $a, b$ with $b$ being a digit
  Output: the numbers q and r resulted from the multiplication (quotient and remainder)
2 let $r[0] \leftarrow 0$
3 for 0 to $i \leftarrow length(a) - 1$: do
4     let $v \leftarrow$ toDecimal($a[i]$) + toDecimal($r[0]$) + toDecimal($base$);
5     $q[i] \leftarrow$ fromDecimal($base, v$ div toDecimal($b[0]$));
6     $r[0] \leftarrow v$ mod toDecimal($b[0]$);
7 end
```

Code Our code becomes:

```
std::pair<Number, Number> Number::operator/(const Number& digit) const {
    if (this->base != digit.base)
        throw std::runtime_error("Operands must be of the same base");

    if (digit.get_value().size() != 1)
        throw std::runtime_error("Division can only be done by DIGIT");

    std::string result_value = "";
    unsigned int division_digit = digitToValue(digit.get_value()[0]);
    unsigned int transport_digit = 0;

    for (int char_index = 0; char_index < this->value_string.size(); char_index++) {
        unsigned int special_value = digitToValue(this->value_string[char_index]) +
                                     transport_digit * this->base;

        transport_digit = special_value % division_digit;
        result_value += valueToDigit(special_value / division_digit);
    }

    return std::make_pair<Number, Number>
        (Number(this->base, result_value),
         Number(this->base, std::string{valueToDigit(transport_digit)}));
}
```

3.3 Conversion Algorithms

3.3.1 Succesive Divsion Method

Pseudocode The successive division method algorithm has the following pseudocode:

Algorithm 5: Successive division conversion algorithm

```
1 function convertSuccessive ($n, src, dst$);
  Input  : the number to be converted $n$,
           the source and destination bases $src$ and $dst$, with $src > dst$
  Output: the converted number in base $dst$, $m$
2 let $q \leftarrow n$
3 while $q \neq 0$ do
4     $pair \leftarrow$ divideNumbers($q$,Number($srcBase$);
5     $q \leftarrow pair.q$;
6     insertDigitAtBeginning($m, pair.r[0]$);
7 end
```

Code Translated to our code base, we get the following implementation:

```cpp
Number convert_successive_division(unsigned int dstBase, const Number& number) {
    if (!(number.base > dstBase))
        throw std::runtime_error(
            "Use the succesive division method only for SrcBase > DstBase");

    const Number base(number.base, std::string{valueToDigit(dstBase)});
    Number quotient(number.base, number.get_value());

    std::string result_string = "";
    while (quotient.get_value() != "0") {
        auto division_pair = quotient / base;

        quotient = division_pair.first;
        result_string.insert(result_string.begin(),
                             division_pair.second.get_value()[0]);
    }

    return Number(dstBase, result_string);
}
```

3.3.2 Substitution Method

Pseudocode The substitution method conversion algorithm has the following pseudocode:

Algorithm 6: Division of a number by a digit (same base)

```
1 function convertSubstitution ($n, src, dst$);
  Input  : the number to be converted $n$,
           the source and destination bases $src$ and $dst$, with $src < dst$
  Output: the converted number in base $dst$, $m$
2 let $baseN \leftarrow$ Number($dstBase$, fromDecimal($dstBase$))
3 let $power \leftarrow$ Number($dstBase$, "1")
4 $m \leftarrow$ Number($dstBase$, "0")
5 for 0 to $i \leftarrow length(a) - 1$: do
6     $m \leftarrow m + power$ * Number($dstBase, n[i]$);
7     $power \leftarrow power * baseN$;
8 end
```

Code Translated to our code base, we get the following implementation:

```cpp
Number convert_substitution(unsigned int dstBase,
                            const Number& number) {
    if (!(number.base < dstBase))
        throw std::runtime_error(
                "Use the substitution method only for SrcBase < DstBase");

    const Number base(dstBase, std::string{valueToDigit(number.base)});
    Number sum(dstBase, "0");
    Number base_power(dstBase, "1");

    const std::string value = number.get_value();
    for (int char_index = value.size() - 1; char_index >= 0; char_index--) {
        sum = sum + base_power * Number(dstBase, std::string{value[char_index]});

        base_power = base_power * base;
    }

    return sum;
}
```

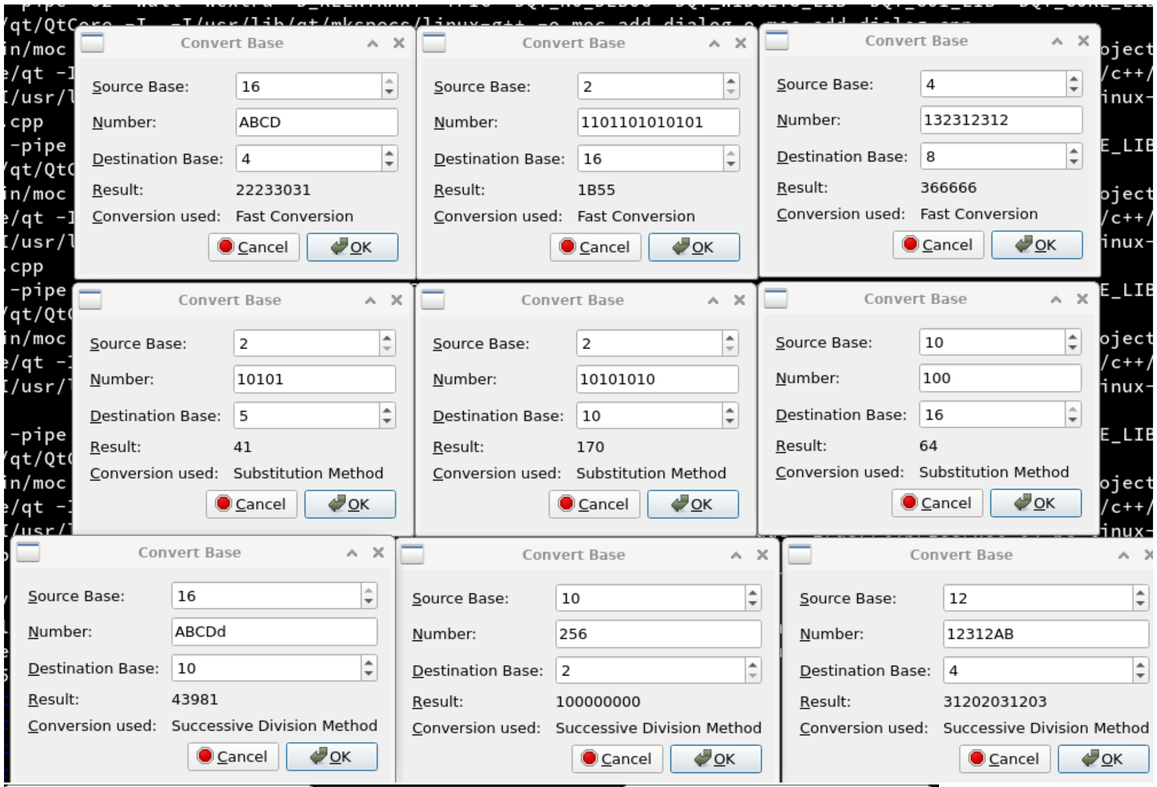
3.3.3 Fast Conversion Method

Code The fast conversion method algorithm is implemented by the following code:

```
Number convert_to_base2(const Number& number) {
    if (number.base == 2)
        return number;

    if (!is_power_of_two(number.base))
        throw std::runtime_error("Use convert_to_base2 only with power of 2 base");

    std::string base2_string = "";
    size_t number_of_digits_used = BINARY_DIGIT_MAX_LENGTH
                                   - get_the_power_of_two(number.base);

    for (const char &character : number.get_value()) {
        std::string base2_digit = RAPID_CONVERSION_STRINGS[digitToValue(character)];

        base2_string += base2_digit.substr(number_of_digits_used, BINARY_DIGIT_MAX_LENGTH);
    }

    return Number(2, base2_string);
}

Number convert_from_base2(unsigned int dstBase, const Number& number) {
    if (dstBase == 2)
        return number;

    if (!isBaseSupported(dstBase))
        throw std::runtime_error("Base not supported");

    if (!is_power_of_two(dstBase))
        throw std::runtime_error("Use convert_to_base2 only with power of 2 base");;

    std::string new_base_string = "";
    const size_t number_of_binary_digits_used = get_the_power_of_two(dstBase);

    std::string base2_string = number.get_value();
    while (base2_string.size() % number_of_binary_digits_used != 0)
        base2_string.insert(base2_string.begin(), '0');

    for (size_t index = 0; index < base2_string.size(); index += number_of_binary_digits_used) {
        std::string base2_digit_string =
            base2_string.substr(index, number_of_binary_digits_used);

        while (base2_digit_string.size() != BINARY_DIGIT_MAX_LENGTH)
            base2_digit_string.insert(base2_digit_string.begin(), '0');

        const size_t digit_value = std::find(RAPID_CONVERSION_STRINGS.cbegin(),
                                             RAPID_CONVERSION_STRINGS.cend(),
                                             base2_digit_string) - RAPID_CONVERSION_STRINGS.cbegin();

        new_base_string += valueToDigit(digit_value);
    }

    return Number(dstBase, new_base_string);
}
```

```
Number convert_fast(unsigned int dstBase, const Number& number) {
    if (!isBaseSupported(dstBase))
        throw std::runtime_error("Base not supported");

    if (!is_power_of_two(dstBase) || !is_power_of_two(number.base))
        throw std::runtime_error("Use convert_to_base2 only with bases that are \
                                  powers of two");
    Number base2_number = convert_to_base2(number);

    return convert_from_base2(dstBase, base2_number);
}
```

4 Test Cases / Examples



Convert Base
Source Base: 16
Number: ABCD
Destination Base: 4
Result: 22233031
Conversion used: Fast Conversion

Convert Base
Source Base: 2
Number: 1101101010101
Destination Base: 16
Result: 1B55
Conversion used: Fast Conversion

Convert Base
Source Base: 4
Number: 132312312
Destination Base: 8
Result: 366666
Conversion used: Fast Conversion

Convert Base
Source Base: 2
Number: 10101
Destination Base: 5
Result: 41
Conversion used: Substitution Method

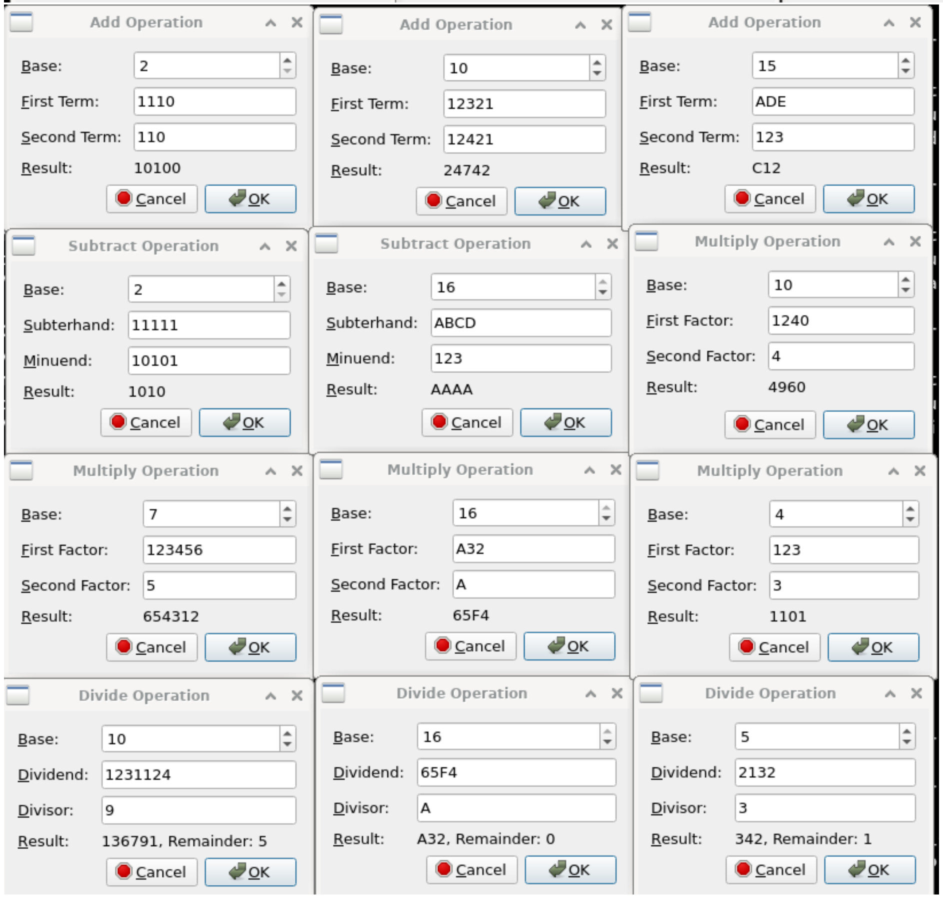
Convert Base
Source Base: 2
Number: 10101010
Destination Base: 10
Result: 170
Conversion used: Substitution Method

Convert Base
Source Base: 10
Number: 100
Destination Base: 16
Result: 64
Conversion used: Substitution Method

Convert Base
Source Base: 16
Number: ABCDd
Destination Base: 10
Result: 43981
Conversion used: Successive Division Method

Convert Base
Source Base: 10
Number: 256
Destination Base: 2
Result: 100000000
Conversion used: Successive Division Method

Convert Base
Source Base: 12
Number: 12312AB
Destination Base: 4
Result: 31202031203
Conversion used: Successive Division Method



Add Operation
Base: 2
First Term: 1110
Second Term: 110
Result: 10100

Add Operation
Base: 10
First Term: 12321
Second Term: 12421
Result: 24742

Add Operation
Base: 15
First Term: ADE
Second Term: 123
Result: C12

Subtract Operation
Base: 2
Subterhand: 11111
Minuend: 10101
Result: 1010

Subtract Operation
Base: 16
Subterhand: ABCD
Minuend: 123
Result: AAAA

Multiply Operation
Base: 10
First Factor: 1240
Second Factor: 4
Result: 4960

Multiply Operation
Base: 7
First Factor: 123456
Second Factor: 5
Result: 654312

Multiply Operation
Base: 16
First Factor: A32
Second Factor: A
Result: 65F4

Multiply Operation
Base: 4
First Factor: 123
Second Factor: 3
Result: 1101

Divide Operation
Base: 10
Dividend: 1231124
Divisor: 9
Result: 136791, Remainder: 5

Divide Operation
Base: 16
Dividend: 65F4
Divisor: A
Result: A32, Remainder: 0

Divide Operation
Base: 5
Dividend: 2132
Divisor: 3
Result: 342, Remainder: 1