DIMPL: An Efficient and Expressive DSL for Discrete Mathematics

Rohit Jha

Department of Computer Science and Engineering
University of California, San Diego

Abstract

This paper describes the language DIMPL, a domain-specific language (DSL) for discrete mathematics. Based on Haskell, DIMPL carries all the advantages of a purely functional programming language. Besides containing a comprehensive library of types and efficient functions covering the areas of logic, set theory, combinatorics, graph theory, number theory and algebra, the DSL also has a notation akin to one used in these fields of study. This paper also demonstrates the benefits of DIMPL by comparing it with C, Fortran, MATLAB and Python —languages that are commonly used in mathematical programming.

Keywords: Discrete mathematics, Programming languages
2010 MSC: 97N70, 68R01, 97P40, 68N15

1. Introduction

Discrete mathematics and several of its fields such as logic and number theory have been studied by humans since ancient times. Modern mathematicians rely on programming languages and expect them to be capable of representing problems and efficiently determining solutions. While programming languages such as C, Fortran, Python and MATLAB are often used by many mathematicians, they may not be the best choice due to the following reasons:

- **Cost of learning:** Mathematicians may not be adept programmers and hence learning new or multiple programming languages could require a fair amount of time and effort.
- **Expressiveness**: The formal representations of discrete structures such as Graphs are not available in existing programming languages. Mathematicians may have to create their own complex data types, which could be unintuitive when using pointers in C.

Email address: rohitjha@ucsd.edu (Rohit Jha)

- **Performance**: The areas of number theory and combinatorics contain many compute-intensive functions and operations. A language such as Python may be convenient for writing programs, but it lags behind many programming languages in terms of performance.

DIMPL (Discrete Mathematics Programming Language) is a domain-specific language (DSL) for discrete mathematics that aims to address these drawbacks through the following ways:

- DIMPL provides a syntax that is identical to the formal notation followed in discrete mathematics, thus being easier to read and grasp [1].
- DIMPL provides infinite-precision integer arithmetic, giving mathematicians the ability to work with signed and unsigned integers that are larger than 64 bits, which is essential in number theory and algebra.
- DIMPL programs are more efficient than many interpreted languages as they are compiled into the target's machine code.

The remaining sections of this paper are organized as follows: section 2 describes the design of DIMPL, section 3 describes a comparative analysis of DIMPL with other languages and section 4 concludes the paper with the future scope of DIMPL.

2. The DIMPL Language

DIMPL is implemented in Haskell, a purely-functional programming language. The following features of Haskell make it an excellent base language:

- **No side-effects:** Haskell is a purely functional language and treats computation as the evaluation of mathematical functions, avoiding state and mutable data [2]. This makes Haskell a good candidate for a base language of a mathematics-oriented DSL.
- **Functions are first-class objects:** Functions in Haskell can be passed to other functions as arguments, can be returned as results from other functions and can also be assigned to variables. This property allows users to work with higher order functions, making programs modular and easier to understand [3].
- **Lazy evaluation:** Lazy evaluation is a part of the operational semantics of Haskell. Haskell evaluations are deferred until their results are required by other computations, allowing programmers to handle infinite data. For example, the Haskell expression `let a = [0 ..]` creates an infinite list of integers beginning at 0. This property is useful while working with infinite sets such as the set of natural numbers $\mathbb{N} = \{0, 1, 2 \ldots\}$ [4].

- **Type system:** Provision of creating algebraic data types makes it convenient to create a DSL in Haskell. Haskell's strong compile-time type checking ensures program reliability [5] and its excellent type inference system helps improve productivity as programmers can use complex types ubiquitously without the type signature.

- **Polymorphic types and functions:** The parametric polymorphism feature in Haskell allows a function to be used with multiple types [6]. For example, a function with type signature as `id :: a -> a` implies that the `id` function can have an argument of `Char`, `Bool`, `Int` or any other type and return a value of the same type. Haskell also supports ad-hoc polymorphism [7], in which a function or operator can perform different operations for different types. For example, the + operator can add two `Int`, `Double` or a user-defined type.

- **Compiled programming language:** The Glasgow Haskell Compiler (GHC) provides a cross-platform environment for Haskell programming with support for numerous libraries and optimizations [8][9][10]. Since GHC compiles Haskell code to native code, the run-time efficiency is much higher than those of interpreted languages such as Python.

- **Smart garbage collection:** Since data is immutable in Haskell, every operation's result needs to be stored in a new value. However, GHC takes advantage of the immutability and clears the older value, hence improving garbage collection when there is a higher percentage of garbage. Besides this, GHC's garbage collector has been tuned to perform efficiently on multi-core machines [11].

2.1. Architecture Design Pattern

To keep the syntax of the language close to the actual formal mathematical representation, DIMPL is implemented as a preprocessed DSL. DIMPL consists of:

1. **Library:** A library of functions and types written in Haskell provides functionality for users to write DIMPL programs easily. The library consists of 12 modules, which are explained in section 2.2.
2. **Preprocessor:** The preprocessor transforms DIMPL code into its equivalent Haskell code. It is a simple script that uses the `sed` (stream editor) utility to parse a DIMPL program and translate it to Haskell.

Figure 1 shows the architecture design pattern of DIMPL. The DIMPL script written by a user is first passed to the preprocessor, which translates the DIMPL code into Haskell. GHC then compiles this Haskell code into the native binary (machine code) by linking the DIMPL and other Haskell libraries. The preprocessing can be performed by passing the `-F -pgmF` options to GHC during compilation.

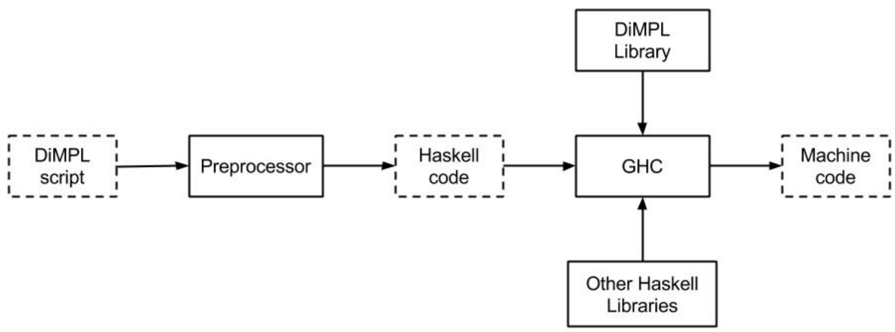

Figure 1: Dimpl architecture design pattern

2.2. Modules

Broadly, Dimpl's library covers the areas of mathematical logic, set theory, linear algebra, combinatorics, number theory and graph theory [12]. The library is divided into 12 modules as shown in Figure 2. The following subsections describe each of the modules in detail.

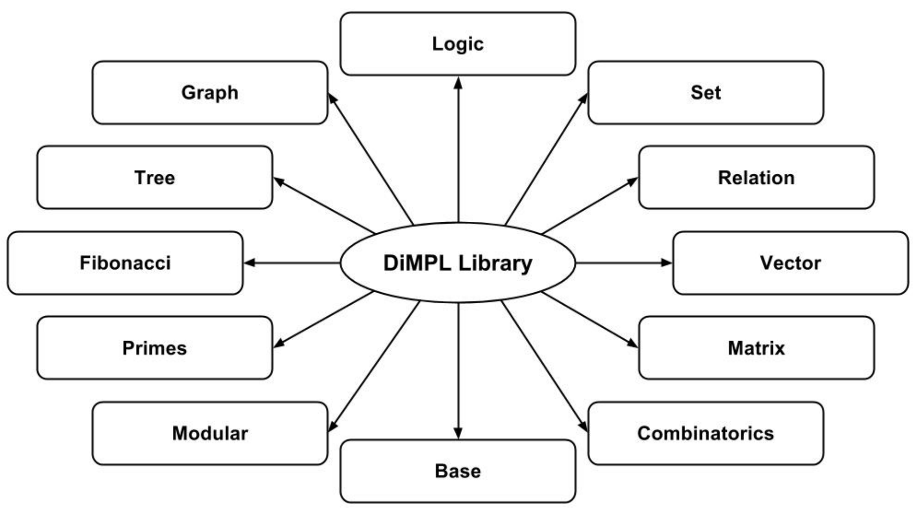

Figure 2: Modules of the Dimpl library

2.2.1. Logic

The Logic module supports first-order logic, which includes logical operators, functions, quantifiers, Boolean algebra and predicate logic.

Dimpl provides syntactic sugar for Haskell operators such as `&&` (AND – logical conjunction) and `||` (OR – logical disjunction) through functions such as `and'` and `or'`. While Haskell has functions named `and` and `or`, they require a list of `Bool` values to be passed as parameter. On the other hand, DIMPL's `and'` and `or'` take just two `Bool` arguments. Additionally, they can be used in their infix form, which is a more intuitive representation.

Since Haskell allows programmers to easily create new binary operators, DIMPL is equipped with operators for conjunction and disjunction. The `/\` and `\/` operators can be used instead of `and'` and `or'`, bringing DIMPL's syntax closer to the actual mathematical notation. DIMPL borrows the logical negation function `not` ($\neg$ or $\sim$), the universal quantifier `forall` ($\forall$) and the existential quantifier `exists` ($\exists$) from Haskell.

Apart from these, DIMPL also adds several other functions and operators for logic. The functions in this module are described in Table 1 and the operators are described in Table 2.

Table 1: Functions in the Logic module

Function	Description	Example
`and'`	Conjunction	`and' True False`
`or'`	Disjunction	`or' True False`
`nand`	Negation of conjunction	`nand True False`
`nor`	Negation of disjunction	`nor True False`
`xor`	Exclusive disjunction	`xor True False`
`xnor`	Negation of XOR	`xnor True False`
`notL`	Negation of a list of `Bool`	`notL [True, False, True]`
`andL`	Conjunction of a list on `Bool`	`andL [True, False, True]`
`orL`	Disjunction of a list of `Bool`	`orL [True, False, True]`
`nandL`	NAND on a list of `Bool`	`nandL [True, False, True]`
`norL`	NOR on a list of `Bool`	`norL [True, False, True]`
`xnorL`	XNOR on a list of `Bool`	`xnorL [True, False, True]`
`equals`	Equality	`equals True False`
`implies`	Implication	`implies True False`

Table 2: Operators in the Logic module

Operator	Description	Example
`/\`	Binary conjunction	`True /\ False`
`\/`	Binary disjunction	`True \/ False`
`==>`	Negation of conjunction	`True ==> False`
`<=>`	Equality	`True <=> False`

Table 3 compares the notation (syntax) of Haskell, DIMPL and mathematical logic. As one can observe, DIMPL would be relatively easier to use for a novice student or mathematician.

Table 3: Comparison of Haskell, DIMPL and mathematical logic notations

<table>
<tr><th>Haskell</th><th>DIMPL</th><th>First-order logic</th></tr>
<tr><td><code>True && False && True</code></td><td><code>True /\ False /\ True</code></td><td>$T \wedge F \wedge T$</td></tr>
<tr><td><code>True || False || True</code></td><td><code>True \/ False \/ True</code></td><td>$T \vee F \vee F$</td></tr>
</table>

2.2.2. Set

The Set module of DIMPL allows users to perform operations on sets of values (numbers, characters, strings, lists, etc.). This module introduces the `Set` type, which internally stores the elements in a list and displays the output with the curly braces surrounding the elements. For example, the DIMPL expression

```
Set {x | x <- natural, x > 10}
```

returns a `Set` of natural numbers greater than 10 – `{11, 12, 13, ...}`. This expression is quite close to the mathematical representation, $\{x \mid x \in \mathbb{N},\ x > 10\}$, and is facilitated by exploiting Haskell's list comprehension.

Table 4 lists all the functions and their brief descriptions. DIMPL functions are usually called using prefix notation and can be called in infix notation by using the `` (backtick) infix operator. For example, to check if the set `vowels = Set {‘a’,‘e’,‘i’,‘o’,‘u’}` is a subset of the set `alphabet = Set {‘a’..‘z’}`, we can call the `isSubset` function as either `isSubset vowels alphabet` or as ``vowels `isSubset` alphabet``. The result returned by both expressions is `True`.

2.2.3. Relation

Using the `Relation` type users can perform operations with binary relations, which are associations between the elements of the domain and the elements of the range. As with `Set`s, `Relation`s can be easily understood and represented in DIMPL since they can be created using Haskell's list comprehension. For example, if $\mathbb{N}$ denotes the set of all natural numbers, then the relation $\leq \subseteq \mathbb{N} \times \mathbb{N}$, which is expressed as $\{(x, y) | x \in \mathbb{N}, y \in \mathbb{N}, x \leq y\}$, can be written in DIMPL as the following expression:

```
Relation {(x,y) | x <- natural, y <- natural, x <= y}
```

which evaluates to `{(1,1),(1,2),(1,3),...}`. All the functions provided by the Relation module are described in Table 5.

2.2.4. Vector

The Vector module introduces the `Vector` type which is used for representing vectors of any order. An example of a third-order vector in DiMPL is `Vector <1,2,0>` and an example of an infinite-order vector is `Vector <1,2,3..>`. This notation of using angle brackets to enclose vector components distinguishes `Vector`s from lists and `Set`s, thereby increasing programs' readability as it is the ordered set notation for vectors in mathematics. For instance, the vector $v = \langle 1, 2, 0\rangle$ is written in DIMPL as `Vector <1,2,0>`.

Table 4: Functions in the Set module

Function	Description
`setToList`	Returns a `Set` as a list
`listToSet`	Returns a list as a `Set`
`union`	Returns the union of two `Set`s
`unionL`	Returns the union of a list of `Set`s
`intersection`	Returns the intersection of two `Set`s
`intersectionL`	Returns the intersection of a list of `Set`s
`difference`	Returns the set difference of two `Set`s
`cardinality`	Returns the number of elements in a `Set`
`powerSet`	Returns the power set of a `Set`
`cartProduct`	Returns the Cartesian product of two `Set`s
`nullSet`	Returns a null `Set`
`natural`	Returns the `Set` of natural numbers $\{1, 2, 3 \ldots\}$
`natural'`	Returns a `Set` of natural numbers up to an upper limit
`whole`	Returns the `Set` of whole numbers $\{0, 1, 2 \ldots\}$
`whole'`	Returns a `Set` of whole numbers up to an upper limit
`isMemberOf`	Checks if a value is an element of a `Set`
`isNotMemberOf`	Checks if a value is not an element of a `Set`
`isNullSet`	Checks if a `Set` is a null set
`isNotNullSet`	Checks if a `Set` is not a null set
`isSubset`	Checks if a `Set` is a subset of another `Set`
`isProperSubset`	Checks if a `Set` is a proper subset of another `Set`
`isSuperset`	Checks if a `Set` is a superset of another `Set`
`areDisjoint`	Checks if two `Set`s are disjoint
`areDisjointL`	Checks if all `Set`s in a list are disjoint
`sMap`	Maps a function to all the elements of a `Set`

Table 6 describes the functions provided by this module. In addition, the Vector module provides syntactic sugar for some of the functions through operators, which are described in Table 7.

To find the volume of a parallelepiped whose three edges are formed by the vectors $u = \langle 3, 2, 1\rangle, v = \langle -1, 3, 0\rangle$ and $w = \langle 2, 2, 5\rangle$, the Dimpl program is:

```
import Vector
let u = Vector <3,2,1>
let v = Vector <(-1),3,0>
let w = Vector <2,2,5>
let volume = stp u v w
```

Another interesting application of the `Vector` type is that it can represent vector functions as well. For example, `f x = Vector <sin x,cos x,tan x>` defines a vector function `f` over one variable, `x`.

Table 5: Functions in the Relation module

Function	Description
`relationToList`	Returns a `Relation` as a list
`listToRelation`	Returns a list as a `Relation`
`inverse`	Returns the inverse of a `Relation`
`getDomain`	Returns the domain of a `Relation`
`getRange`	Returns the range of a `Relation`
`elements`	Returns the elements of a `Relation`
`returnDomainElems`	Returns the elements of a `Relation`'s domain
`returnRangeElems`	Returns the elements of a `Relation`'s range
`rUnion`	Returns the union of two `Relations`
`rUnionL`	Returns the union of `Relations` in a list
`rIntersection`	Returns the intersection of two `Relations`
`rIntersectionL`	Returns the intersection of `Relations` in a list
`rDifference`	Returns the difference of two `Relations`
`rComposite`	Returns the concatenation of two `Relations`
`rPower`	Returns the n^{th} power of a `Relation`
`reflClosure`	Returns the reflexive closure of a `Relation`
`symmClosure`	Returns the symmetric closure of a `Relation`
`tranClosure`	Returns the transitive closure of a `Relation`
`isReflexive`	Checks if a `Relation` is reflexive
`isIrreflexive`	Checks if a `Relation` is irreflexive
`isSymmetric`	Checks if a `Relation` is symmetric
`isAsymmetric`	Checks if a `Relation` is asymmetric
`isAntiSymmetric`	Checks if a `Relation` is anti-symmetric
`isTransitive`	Checks if a `Relation` is transitive
`isEquivalent`	Checks if a `Relation` is equivalent
`isWeakPartialOrder`	Checks if a `Relation` is a weak partial order
`isWeakTotalOrder`	Checks if a `Relation` is a weak total order
`isStrictPartialOrder`	Checks if a `Relation` is a strict partial order
`isStrictTotalOrder`	Checks if a `Relation` is a strict total order

2.2.5. Matrix

The `Matrix` type introduced in this module allows users to work with matrices. The `Matrix` type internally stores values as a two-dimensional list. For example, consider the third-order unit/identity matrix – `Matrix [[1,0,0], [0,1,0], [0,0,1]]` – which is displayed as:

```
1  0  0
0  1  0
0  0  1
```

This allows users to easily comprehend results of matrix operations. Dimpl's Matrix module contains numerous functions for handling matrices and these are

Table 6: Functions in the Vector module

Function	Description
`dimension`	Returns the dimension/order of a `Vector`
`magnitude`	Returns the magnitude of a `Vector`
`vectorToList`	Returns a `Vector` as a list
`listToVector`	Returns a list as a `Vector`
`vAdd`	Returns the sum of two `Vectors`
`vAddL`	Returns the sum of `Vectors` in a list
`vSub`	Returns the difference of two `Vectors`
`vSubL`	Returns the difference of `Vectors` in a list
`innerProd`	Returns the inner product of two `Vectors`
`angle`	Returns the angle/direction of a `Vector`
`scalarMult`	Returns the product of a scalar and a `Vector`
`crossProduct`	Returns the cross product of two `Vectors`
`stp`	Returns the scalar triple product of three `Vectors`
`vtp`	Returns the vector triple product of three `Vectors`
`extract`	Returns the component at a particular index
`extractRange`	Returns the components at a range of indices
`vMap`	Maps a function to all components of a `Vector`
`normalize`	Returns the normalized form of a `Vector`
`isNullVector`	Checks if a `Vector` is a null vector
`areOrthogonal`	Checks if two `Vectors` are orthogonal

Table 7: Operators in the Vector module

Operator	Description	Example
`<+>`	Add two `Vectors`	`Vector <1,2> <+> Vector <0,1>`
`<->`	Subtract two `Vectors`	`Vector <1,0> <-> Vector <(-2)>`
`<.>`	Dot product	`Vector <4,6> <.> Vector <1,5.5>`
`<*>`	Scalar multiplication	`3 <*> Vector <2.5,0,(-1.7)>`
`><`	Cross product	`Vector <1,1> >< Vector <2,3>`

listed in Table 8. This module also provides five `Matrix` operators for common matrix operations – addition, subtraction, division and multiplication (with a scalar or `Matrix`). These are described in Table 9 with examples. In all the examples, `m1` and `m2` have type `Matrix`.

Provision of a number of functions allows users to easily write programs. For example, using the `inverse` function and the matrix multiplication operator `|><|`, a function to solve linear equations of any order can be written as:

```
import Matrix
solveEqns ::  Num a => Matrix a -> Matrix a -> Matrix a
solveEqns (Matrix coeff) (Matrix const)
= inverse (Matrix coeff) |><| Matrix const
```

Table 8: Functions in the Matrix module

Function	Description
mAdd	Returns the sum of two Matrix
mAddL	Returns the sum of a list of Matrix
mSub	Returns the difference of two Matrix
mSubL	Returns the difference of a list of Matrix
transpose	Returns the transpose of a Matrix
mScalarMult	Returns the product of a scalar and a Matrix
mMult	Returns the product of two Matrix
mMultL	Returns the product of a list of Matrix
numRows	Returns the number of rows in a Matrix
numCols	Returns the number of columns in a Matrix
matrixToList	Returns a Matrix as a two-dimensional list
listToMatrix	Returns a two-dimensional list as a Matrix
determinant	Returns the determinant of a Matrix
inverse	Returns the inverse of a Matrix
mDiv	Returns the result of dividing two Matrix
extractRow	Returns a particular row of a Matrix
extractCol	Returns a particular column of a Matrix
extractRowRange	Returns a list of rows of a Matrix
extractColRange	Returns a list of columns of a Matrix
mPower	Returns a Matrix raised to the n^{th} power
trace	Returns the trace of a Matrix
isInvertible	Checks if a Matrix is invertible
isSymmetric	Checks if a Matrix is symmetric
isSkewSymmetric	Checks if a Matrix is skew-symmetric
isRow	Checks if a Matrix is a row matrix
isColumn	Checks if a Matrix is a column matrix
isSquare	Checks if a Matrix is a square matrix
isOrthogonal	Checks if a Matrix is orthogonal
isInvolutive	Checks if a Matrix is involutive
isZeroOne	Checks if a Matrix is a zero one matrix
isZero	Checks if a Matrix is a zero matrix
isOne	Checks if a Matrix is a one matrix
isUnit	Checks if a Matrix is a unit/identity matrix
mMap	Maps a function to a Matrix
zero	Returns a square zero matrix of the order mentioned
zero'	Returns a $M \times N$ zero matrix
one	Returns a square one matrix of the order mentioned
one'	Returns a $M \times N$ one matrix
unit	Returns a unit/identity matrix of the order mentioned

Consider a system of two simultaneous linear equations:

$$2x - 3y = -2$$
$$4x + y = 24$$

This system of equations can be represented using matrices as:

$$\begin{bmatrix} 2 & -3 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -2 \\ 24 \end{bmatrix}$$

The `solveEqns` function can be called as follows to solve this system of equations:

```
m1 = Matrix [[2,(-3)], [4,1]]
m2 = Matrix [[(-2)],[24]]
result = solveEqns m1 m2
```

The `result` matrix then has the value `Matrix [[5],[4]]`, which indicates that the solution of the two equations is $x = 5$ and $y = 4$.

Table 9: Operators in the Matrix module

Operator	Description	Example
`|+|`	Add two `Matrix`	`m1 |+| m2`
`|-|`	Subtract two `Matrix`	`m1 |-| m2`
`|/|`	Divide two `Matrix`	`m1 |/| m2`
`|*|`	Multiply a scalar and a `Matrix`	`42 |*| m1`
`|><|`	Multiply two `Matrix`	`m1 |><| m2`

2.2.6. Combinatorics

The Combinatorics module in DIMPL provides functions to calculate factorial, determine the number of permutations and combinations, find all possible permutations and combinations, and shuffle a list of values. The functions for these are listed in Table 10

Table 10: Functions in the Combinatorics module

Function	Description
`factorial`	Returns the factorial of an `Integer`
`p`	Returns the number of possible permutations of a list
`c`	Returns the number of possible combinations of a list
`permutation`	Returns a list of all permuations
`combination`	Returns a list of all combinatons
`shuffle`	Returns a list after shuffling all its elements

Since finding the number of possible permutations and combinations may involve calculating the factorial of large numbers, the `factorial` function supports infinite-precision integers. Furthermore, the functions `p` and `c` have been optimized; instead of naively implementing the functions as

```
p = factorial n `div` factorial (n - r)
c = factorial n `div`(factorial r * factorial (n - r))
```

the functions have been implemented as

```
p ::  Integer -> Integer -> Integer
p n r
  | n < 1 = error "Usage - p n r, where 'n' is positive."
  | r < 1 = error "Usage - p n r, where 'r' is positive."
  | otherwise = product [(a-b+1) ..  a]
    where
      a = max n r
      b = min n r

c ::  Integer -> Integer -> Integer
c n r
  | n < 1 = error "Usage - c n r, where 'n' is positive."
  | r < 1 = error "Usage - c n r, where 'r' is positive."
  | otherwise = product [(b+1) ..  n] `div` product [1 ..  (a-b)]
    where
      a = max n r
      b = min n r
```

2.2.7. Base

The Base module provides functions that allow users to convert numbers between various number bases. These are described in Table 11.

Table 11: Functions in the Base module

Function	Description
`toBase`	Returns the equivalent of a decimal number in another base
`toBin`	Returns a decimal number in its binary equivalent
`toOct`	Returns a decimal number in its octal equivalent
`toHex`	Returns a decimal number in its hexadecimal equivalent
`fromBase`	Returns the decimal equivalent of a number in any base
`fromBin`	Returns the decimal equivalent of a binary number
`fromOct`	Returns the decimal equivalent of an octal number
`fromHex`	Returns the decimal equivalent of a hexadecimal number
`toAlpha`	Returns a number in its equivalent alphanumeric form
`fromAlpha`	Returns a number from its equivalent alphanumeric form

A notable feature of the functions converting decimal integers to other bases is that the result is returned as a list of digits. For example, `toBin 32` returns `[1,0,0,0,0,0]`. This allows us to handle bases such as the sexagecimal (base 60), in which a single digit may actually be equivalent to two-digits in decimal. Thus, the sexagesimal equivalent of 10,000 is returned by the expression `toBase 60 10000`, which evaluates to `[2,46,40]`.

2.2.8. Modular

Modular arithmetic is widely used in cryptography [13][14] and computer algebra. The Modular module allows users to perform basic modular arithmetic operations and solve congruence relations using the functions described in Table 12.

Table 12: Functions in the Modular module

Function	Description
`modAdd`	Returns sum using modular arithmetic
`modSub`	Returns difference using modular arithmetic
`modMult`	Returns product using modular arithmetic
`modExp`	Returns result after modular exponentiation
`isCongruent`	Checks for modular congruency $a \equiv b \pmod{n}$
`findCoungruentPair`	Returns x from $ax \equiv b \pmod{n}$
`findCoungruentPair'`	Returns x from $a + x \equiv b \pmod{n}$

The `modExp` function can be used to demonstrate a trivial working program of the Diffie-Hellman key exchange protocol. According to the protocol [15], if α is a primitive root and q is a prime number, both of which are known to both Alice and Bob (the two communicating parties), then Alice's and Bob's public keys are given by

$$Y_A = \alpha^{X_A} \bmod q$$
$$Y_B = \alpha^{X_B} \bmod q$$

where X_A and X_B are their respective private keys. Alice can calculate the shared key K_A as

$$\begin{aligned} K_A &= (Y_B)^{X_A} \bmod q \\ &= \alpha^{X_B X_A} \bmod q \end{aligned}$$

Bob can calculate the shared key K_B as

$$\begin{aligned} K_B &= (Y_A)^{X_B} \bmod q \\ &= \alpha^{X_A X_B} \bmod q \\ &= \alpha^{X_B X_A} \bmod q \\ &= K_A \end{aligned}$$

Using DIMPL's Modular library module, the program can be written as:

```
import Modular

publicKey ::  Integer -> Integer -> Integer -> Integer
publicKey a b c = modExp a b c

sharedKey ::  Integer -> Integer -> Integer -> Integer
sharedKey pubKey priKey prime = modExp pubKey priKey prime

q = 23   -- prime number
alpha = 5   -- primitive root
xA = 6   -- Alice's private key
xB = 15   -- Bob's private key

yA = publicKey alpha xA q   -- (5 ^ 6) mod 23 = 8
yB = publicKey alpha xB q   -- (5 ^ 15) mod 23 = 19
kA = sharedKey yB xA q   -- (19 ^ 6) mod 23 = 2
kB = sharedKey yA xB q   -- (8 ^ 15) mod 23 = 2
```

2.2.9. Primes

The Primes module in DIMPL's library provides functions for efficient generation of prime numbers, primality testing and prime factorization. These allow users in the field of cryptography to develop efficient programs for encryption and also for cryptanalysis [13][14]. All the functions of this module are described in Table 13.

Table 13: Functions in the Primes module

Function	Description
`primesTo`	Returns all primes less than a specified limit
`primesBetween`	Returns all primes between two numbers
`nPrimes`	Returns the first n primes
`isPrime`	Checks if an `Integer` is a prime number
`nextPrime`	Returns a prime $\geq$ to an `Integer`
`primeFactors`	Returns all the prime factors of an `Integer`
`uniquePrimeFactors`	Returns all the unique prime factors of an `Integer`
`areCoprime`	Checks if two `Integers` are coprime

Since the Primes module has efficient implementations of `primesTo` and `isPrime`, these functions can be applied in the calculation of Mersenne prime numbers. A Mersenne prime number M_p is a prime number of the form $2^p - 1$, where p is also a prime number. A simple program in DIMPL to generate the set of all values of p is

```
import Set
import Primes

mersennePrimePowersTo n
= Set { p | p <- primesTo n, isPrime (2^p - 1) }
```

When invoked as `mersennePrimePowersTo 2000`, the result returned is `Set {2,3,5,7,13,17,19,31,61,89,107,127,521,607,1279}`.

2.2.10. Fibonacci

The Fibonacci sequence is a sequence of number where a term is the sum of the last two terms, 0 and 1 being the first two terms. This module can be applied in the Fibonacci search algorithm, Fibonacci heap data structure and Fibonacci cube graphs for interconnecting parallel and distributed systems [16][17]. The functions of this module are described in Table 14.

Table 14: Functions in the Fibonacci module

Function	Description
`fib`	Returns the n^{th} term in the Fibonacci sequence
`fibSeq`	Returns the first n terms in the Fibonacci sequence
`fibIndex`	Returns the index of a term in the Fibonacci sequence
`isFibNum`	Checks if an `Integer` is a term of the Fibonacci sequence

The `fib`, `fibSeq` and `fibIndex` functions take an `Integer` as argument and return an `Integer`, thus handling numbers larger than $2^{63} - 1$.

2.2.11. Tree

The `BinTree` data type in the Tree module represents a binary tree and is defined internally as:

```
data BinTree a =
  Leaf | Node a (BinTree a) (BinTree a) deriving (Eq, Show)
```

A `BinTree` can consist of either a `Leaf` (leaf node) or a `Node` with a value and two `BinTree` children. For example, the following statement defines `tree` as a `BinTree` holding values from 1 through 8:

```
let tree =
  Node 4
    (Node 2
      (Node 1 Leaf Leaf) (Node 3 Leaf Leaf))
    (Node 7
      (Node 5 Leaf (Node 6 Leaf Leaf))
      (Node 8 Leaf Leaf))
```

Table 15: Functions in the Tree module

Function	Description
`inorder`	Returns nodes of a `BinTree` in inorder sequence
`preorder`	Returns nodes of a `BinTree` in preorder sequence
`postorder`	Returns nodes of a `BinTree` in postorder sequence
`singleton`	Returns a singleton `Node`
`addNode`	Returns a `BinTree` after adding a `Node` to a `BinTree`
`hasValue`	Checks if a `BinTree` has a `Node` of a specified value
`reflect`	Returns the mirror-reflection of a `BinTree`
`height`	Returns the height of a `BinTree`
`depth`	Returns the depth of a `Node`
`size`	Returns the number of `Nodes` in a `BinTree`
`isBalanced`	Checks if a `BinTree` is a balanced binary Tree

This is the Dimpl representation of the binary tree shown in Figure 3. Table 15 describes the functions provided by the Tree module to work with the `BinTree` data type.

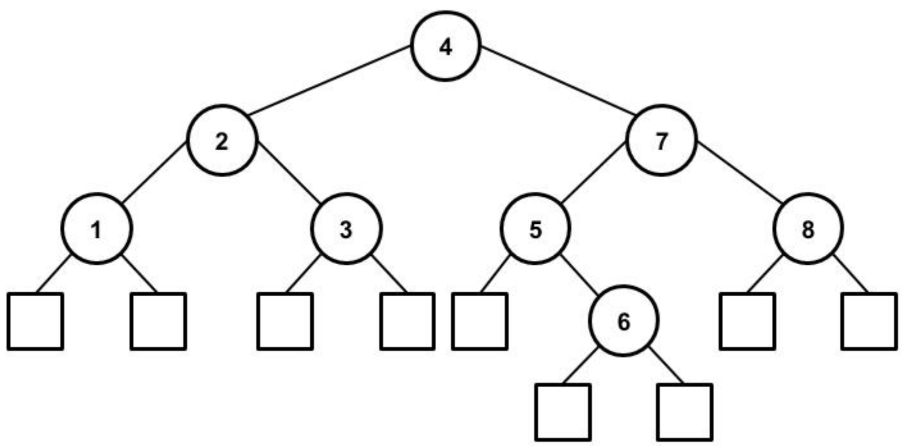

Figure 3: A binary tree with nodes numbered 1 through 8. The square nodes represent leaf nodes of this binary tree.

2.2.12. Graph

In discrete mathematics, graphs are denoted by $G(V, E)$, where V is the set of vertices and E is the set of edges [18]. Graphs in Dimpl have a representation exactly like their mathematical notation; the `Graph` type combines the `Vertices` and `Edges` types to represent graphs. The `Vertices` type is a `Set` of all vertices in a `Graph` and the `Edges` type is a `Set` of tuples containing the starting vertex, ending vertex and the weight of the edge connecting these vertices. Thus, if we have a set of vertices `v = Vertices {1,2,3}`, a set of edges

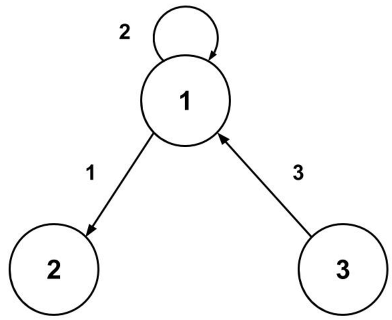

Figure 4: A directed graph with three vertices

e = Edges {(1,2,1),(1,1,2),(3,1,3)}, then a Graph can be defined as let g = Graph (v,e). When printed, this Graph is displayed as:

```
Graph ({1,2,3}, {(1,2,1),(1,1,2),(3,1,3)})
```

This graph can be visualized as the one shown in Figure 4. Since graphs can also be represented as matrices (adjacency matrices with the matrix elements holding edge weights) to easily solve certain problems, the Graph module provides the GraphMatrix type. A GraphMatrix is simply a two-dimensional list and is defined as newtype GraphMatrix a = GraphMatrix [[a]] deriving (Eq). The Graph g defined above can be represented as a GraphMatrix as: GraphMatrix [[0,1,0], [2,0,0], [3,0,0]] and is displayed like a Matrix:

```
1  0  0
0  2  0
3  0  0
```

Table 16: Functions in the Graph module

Function	Description
verticesToList	Returns Vertices as a list
listToVertices	Returns a list as Vertices
edgesToList	Returns Edges as a list
listToEdges	Returns a list as Edges
graphToMatrix	Returns a Graph as a Matrix
matrixToGraph	Returns a Matrix as a Graph
getVerticesG	Returns all Vertices of a Graph
getVerticesGM	Returns all Vertices of a GraphMatrix
numVerticesG	Returns the number of Vertices in a Graph
numVerticesGM	Returns the number of Vertices in a GraphMatrix

`getEdgesG`	Returns all `Edges` in a `Graph`
`getEdgesGM`	Returns all `Edges` in a `GraphMatrix`
`numEdgesG`	Returns the number of `Edges` in a `Graph`
`numEdgesGM`	Returns the number of `Edges` in a `GraphMatrix`
`convertGM2G`	Returns a `GraphMatrix` as a `Graph`
`convertG2GM`	Returns a `Graph` as a `GraphMatrix`
`transposeG`	Returns the transpose of a `Graph`
`transposeGM`	Returns the transpose of a `GraphMatrix`
`isDirectedG`	Checks if a `Graph` is directed
`isDirectedGM`	Checks if a `GraphMatrix` is directed
`isUndirectedG`	Checks if a `Graph` is undirected
`isUndirectedGM`	Checks if a `GraphMatrix` is undirected
`unionG`	Returns the union of two `Graph`
`unionGM`	Returns the union of two `GraphMatrix`
`addVerticesG`	Adds `Vertices` to a `Graph`
`addVerticesGM`	Adds `Vertices` to a `GraphMatrix`
`getVerticesFromEdges`	Returns `Vertices` in `Edges`
`addEdgesG`	Adds `Edges` to a `Graph`
`addEdgesGM`	Adds `Edges` to a `GraphMatrix`
`areConnectedGM`	Checks if `Vertices` are connected in a `GraphMatrix`
`numPathsBetweenGM`	Returns the number of paths between two `Vertices`
`adjacentNodesG`	Returns adjacent nodes of a `Graph`'s vertex
`adjacentNodesGM`	Returns adjacent nodes of a `GraphMatrix`'s vertex
`inDegreeG`	Returns the in-degree of a `Graph`'s vertex
`inDegreeGM`	Returns the in-degree of a `GraphMatrix`'s vertex
`outDegreeG`	Returns the out-degree of a `Graph`'s vertex
`outDegreeGM`	Returns the out-degree of a `GraphMatrix`'s vertex
`degreeG`	Returns the degree of a `Graph`'s vertex
`degreeGM`	Returns the degree of a `GraphMatrix`'s vertex
`countOddDegreeV`	Returns the number of vertices with odd degree
`countEvenDegreeV`	Returns the number of vertices with even degree
`isSubgraphG`	Checks if a `Graph` is a subgraph of another
`isSubgraphGM`	Checks if a `GraphMatrix` is a subgraph of another

`hasEulerCircuitG`	Checks if a `Graph` has a Euler Circuit
`hasEulerCircuitGM`	Checks if a `GraphMatrix` has a Euler Circuit
`hasEulerPathG`	Checks if a `Graph` has a Euler Path
`hasEulerPathGM`	Checks if a `GraphMatrix` has a Euler Path
`hasEulerPathNotCircuitG`	Checks if a `Graph` has a Euler Path not Circuit
`hasEulerPathNotCircuitGM`	Checks if a `GraphMatrix` has a Euler Path not Circuit
`hasHamiltonianCircuitG`	Checks if a `Graph` has a Hamiltonian Circuit
`hasHamiltonianCircuitGM`	Checks if a `GraphMatrix` has a Hamiltonian Circuit

Table 16 describes all the functions provided by the Graph module. Most of the functions are applied to graphs in either the `Graph` or the `GraphMatrix` types; the functions with a suffix 'G' are applied to the `Graph` type and the functions with suffix 'GM' are applied to the `GraphMatrix` type.

3. Comparative Analysis

The primary objective of creating a DSL is to provide a clear and efficient representation of problems and solutions. This section compares DIMPL with C, Fortran, MATLAB and Python as these four languages are often used in mathematical programming.

3.1. Programming Paradigm

Programming paradigms dictate how programmers construct solutions to problems. For example, with a functional programming language, programs are constructed using functions, whereas with object-oriented programming, programs are built up using classes and methods. Other common paradigms include imperative, procedural, logic, generic and reflective paradigms. Table 17 shows a comparison of various programming paradigms supported by the five languages under consideration.

Table 17: Comparison of programming paradigms supported

Language	Functional	Object-oriented	Procedural	Generic	Reflective
DIMPL	✓			✓	✓
C			✓		
Fortran		✓	✓	✓	
MATLAB		✓	✓		
Python	✓	✓		✓	

For representing solutions in discrete mathematics, a programming language should ideally treat functions as first-class objects. This also allows users to work with higher-order functions and promotes modularity. As we can see in

Table 17, the languages offering this are Dimpl and Python. Furthermore, mathematical functions and values must not be mutable. As Dimpl is a purely functional programming language, it does not allow for side-effects [2], giving it an advantage over Python. Since Dimpl inherits parametric polymorphism from Haskell [6], a function can be applied to multiple types of parameters without multiple definitions (generic programming), making programs concise and easier to comprehend. On the other hand, Python does not support the generic programming paradigm and programmers are responsible for writing programs to handle multiple types. Thus, from a programming paradigms perspective, Dimpl is the best language for discrete mathematics.

3.2. Type Safety and Type Checking

In a type-safe language, it is guaranteed that the value of an expression will be a proper member of the expression's type. The C programming language is type-unsafe since it allows copying data to a type even when the data may not be of that type [19]. For example, it is possible to copy data between two pointers of different types, violating type and memory safety.

Type-checking refers to the method by which the types of expressions in a program are checked. This checking can be done during compile-time (static) or during run-time (dynamic). Since static type checking is done on a program's code, it can usually handle bugs earlier. Also, in type-safe languages static type checking can improve the efficiency of binary generated by skipping dynamic safety checks [20]. In a language handling types such as Sets, Relations, Vectors, Matrices, Graphs, Edges and Vertices, it becomes essential to check for the type at compile-time as internally all these types may be represented by arrays or lists.

Table 18: Type safety, type checking and infinite-precision support

Language	Type safe	Static type checking	Infinite-precision
Dimpl	✓	✓	✓
C		✓	
Fortran	✓	✓	
MATLAB	✓		
Python	✓	✓	✓

Table 18 compares the type safety and type checking that is performed on the five languages under consideration. A type-safe and statically-typed language would be beneficial due to the reasons mentioned above, which means Dimpl, Fortran and Python are good candidates for a language involving operations on discrete structures.

3.3. Infinite-Precision Arithmetic

With infinite-precision arithmetic, digits of precision do not depend on the register size of a processor, but only on the available memory. For instance, on a 64-bit computer the largest signed integer that can be stored in a register is

$2^{64} - 1 = 18,446,744,073,709,551,615$. While this may seem sufficiently large, several application areas of discrete mathematics such as cryptography require handling of larger numbers.

Table 18 also compares the five languages to show which of them support infinite-precision integer arithmetic without any additional packages. DIMPL's `Integer` type and Python's `int`, provide functionality to work with integers larger than 64 bits.

3.4. Run-time Efficiency

An important factor for selecting a programming language is the efficiency of programs written in that language. While efficiency can be measured on the basis of memory footprint or the number of lines of code (LOC), this paper compares only the run-time efficiency of DIMPL with C, Fortran, MATLAB and Python through five compute-intensive operations.

All programs were executed on a machine having the following configuration:

- CPU: Intel Core i7-5500U (2.40 GHz)
- L1 cache: 128 KB
- L2 cache: 512 KB
- L3 cache: 4.00 MB
- Memory: 8 GB DDR3 (1600 MHz)
- Operating System: Ubuntu 15.10 (x86_64) with Linux kernel 4.2.0
- Haskell Compiler: Glasgow Haskell Compiler (GHC) 7.8.4
- C Compiler: GCC 5.2.1
- Fortran Compiler: GNU Fortran 5.0
- MATLAB version: 2012b
- Python version: 3.4.3

The programs were compiled with the optimization flags enabled for the respective compilers – `-O2` flag for GHC and `-O3` flag for both GCC and GNU Fortran. The run-times of the programs were calculated by using the `perf stat -r 100 <program>` command, which executes '`program`' 100 times and returns the mean running time.

Table 19: Running time of `fibonacci(n)` in milliseconds

Language	$n = 1$	$n = 20$	$n = 40$	$n = 60$	$n = 80$	$n = 100$
DIMPL	1.107	1.160	1.181	1.208	1.248	1.281
C*	0.744	0.754	0.783	0.797	0.813	–
C	0.754	0.979	427.082	212678.42	$>10^7$	–
Fortran*	1.250	1.313	1.411	1.449	1.502	–
Fortran	1.135	1.463	458.636	531654.15	$>10^7$	–
MATLAB	0.211	353.280	612572.81	$>10^7$	$>10^7$	–
Python*	28.912	29.034	29.055	29.129	29.178	29.342
Python	28.392	34.403	55190.385	$>10^7$	$>10^7$	$>10^7$

3.4.1. Generating Fibonacci Terms

The Fibonacci sequence is defined by the recurrence relation $F_n = F_{n-1} + F_{n-2}$, with $F_0 = 0$ and $F_1 = 1$. Table 19 compares the running time of the `fibonacci(n)` function which generates the n^{th} Fibonacci term. Note that the running time for languages marked with a '*' represent the running time of the iterative implementation of the function.

From the table, we can infer that DIMPL is only behind C in terms of run-time efficiency of the `fibonacci(n)` implementation, which is not surprising given the maturity of GCC. However, since DIMPL supports infinite-precision arithmetic, it can generate larger terms. Another interesting observation from Table 19 is that recursive implementations of the `fibonacci(n)` function in C, Fortan and Python have a significantly longer running time, even exceeding 1,000 seconds for generating the 80^{th} Fibonacci term.

3.4.2. Calculating Factorial

The factorial of a positive integer n, denoted by $n!$, is defined as the product of all positive integers less than or equal to n, with $0! = 1$. The factorial operation is heavily applied in combinatorics, algebra and mathematical analysis and is hence chosen as a benchmark to compare the run-time efficiency of the five languages. Table 20 compares the running time of the `factorial(n)` function, which returns the factorial of the integer n.

Table 20: Running time of `factorial(n)` in milliseconds

Language	$n = 1$	$n = 20$	$n = 50$	$n = 100$	$n = 150$	$n = 200$
DIMPL	1.107	1.160	1.181	1.208	1.248	1.281
C*	0.744	0.754	0.783	0.797	0.813	–
C	0.754	0.979	427.082	212678.42	$>10^7$	–
Fortran*	1.250	1.313	1.411	1.449	1.502	–
Fortran	1.135	1.463	458.636	531654.15	$>10^7$	–
MATLAB	0.211	353.280	612572.81	$>10^7$	$>10^7$	–
Python*	28.912	29.034	29.055	29.129	29.178	29.342
Python	28.392	34.403	55190.385	$>10^7$	$>10^7$	$>10^7$

The table indicates that MATLAB's built-in `factorial` function has the least running time, but with the inability to calculate factorials when $n > 171$. Thus, DiMPL's factorial function compensates for its relatively slower run-time (behind MATLAB and C) with its ability to calculate factorials of larger numbers.

3.4.3. Generating Prime Numbers

The third test for measuring run-time efficiency is to generate all prime numbers up to n through the function `primes(n)`. Table 21 depicts the running time of `primes(n)` across the five languages. The function was implemented using the Sieve of Eratosthenes prime sieve algorithm in C, Fortran and Python.

Table 21: Running time of `primes(n)` in milliseconds

Language	$n = 10^3$	$n = 10^4$	$n = 10^5$	$n = 10^6$	$n = 10^7$
DIMPL	1.793	4.193	7.318	16.083	113.87
C	0.971	1.903	3.875	30.728	363.03
Fortran	1.863	3.416	8.428	40.32	416.41
MATLAB	3.109	3.653	3.981	15.609	99.206
Python	35.289	36.697	78.281	398.64	3985.2

For $n \leq 10^4$, `primes(n)` is most efficient in C, followed by Fortran. For $n \geq 10^7$, MATLAB's built-in function `primes` proves to be the most efficient, followed by DiMPL's `primesTo` function from the Primes module. Another interesting observation is that the running time increases drastically in all languages when $n > 10^5$.

3.4.4. Factorizing Integers

Prime factorization of integers is one of the most computationally difficult problems and this property is widely applied in cryptography. For example, factoring a 232-digit number (RSA-768) took two years while hundreds of computers were used [21]. In Table 22, the running time of the `primeFactors` function from DIMPL's Primes module is compared with MATLAB's `factorize` and a trial-division based implementation in C, Fortran and Python for all integers up to n. In this benchmark, all numbers less than or equal to the input were factorized.

Table 22: Running time of `factorize(n)` in milliseconds

Language	$n = 10$	$n = 10^2$	$n = 10^3$	$n = 10^4$	$n = 10^5$	$n = 10^6$
DIMPL	1.184	1.611	2.487	11.947	160.56	1524.8
C	0.820	0.848	1.583	9.794	115.99	1216.3
Fortran	1.251	1.899	3.428	13.084	179.28	1929.4
MATLAB	3.532	3.939	37.210	425.83	381.38	3626.5
Python	30.369	30.517	36.900	151.74	2425.9	31522

As one can observe, the language with the most efficient prime factorization is C, followed by DIMPL.

3.4.5. Solving Simultaneous Linear Equations

Using matrix multiplication and matrix inversion, it is possible to solve simultaneous linear equations in n variables. If we represent all the coefficients in a $n \times n$ coefficient matrix A, all the constants in a constant column matrix B of order n and if X represents the variable column matrix of order n, then the equations can be represented by $A.X = B$. Thus, $X = B.A^{-1}$, implying that the variables can be calculated by multiplying the constant matrix and the inverse of the coefficient matrix. The DIMPL program for this is shown as an example in Section 2.2.5 and it is compared with equivalent implementations in C, Fortran, MATLAB and Python.

Table 23: Running time of `solve(n)` in milliseconds

Language	$n = 10$	$n = 20$	$n = 40$	$n = 60$	$n = 80$	$n = 100$
DIMPL	2.129	3.814	15.652	47.890	326.24	681.66
C	1.319	3.961	16.711	50.319	370.99	745.10
Fortran	2.065	4.227	18.516	53.743	425.88	791.61
MATLAB	1.140	2.933	15.006	46.150	313.72	688.23
Python	39.583	60.860	92.887	259.21	1050.6	12651

Table 23 shows the comparison of running time of the programs implementing the `solve(n)` function, where n is the number of simultaneous linear equations. For $n < 100$, DIMPL loses out only to MATLAB in terms of run-time efficiency but has the least running time for $n = 100$.

4. Conclusions and Future Scope

This paper introduces DIMPL, a preprocessed domain-specific language for discrete mathematics based on Haskell, and covers its various modules. DIMPL offers a syntax that is close to the mathematical notations, making it an ideal choice not only for users working in the application fields such as cryptography and image processing, but also for teaching introductory discrete mathematics to students. This paper also compares the features and performance of DIMPL with C, Fortran, MATLAB and Python to demonstrate how DIMPL stands favorably against them, besides having high expressiveness and a better runtime efficiency in many cases.

Future versions of DIMPL will have an extended library comprising of modules for lattices, groups, rings, monoids and other discrete structures. They will also contain additional functions for the existing modules such as Graph and Tree. Moreover, incorporating Haskell's support for pure parallelism and explicit concurrency in the library functions could significantly improve the efficiency of some functions on multi-core machines.

References

[1] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys (CSUR)*, 37(4):316–334, 2005.

[2] Benjamin Goldberg. Functional programming languages. *ACM Computing Surveys (CSUR)*, 28(1):249–251, 1996.

[3] John Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–107, 1989.

[4] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A History of Haskell: being lazy with class. In *Proceedings of The Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*, pages 1–55, 2007.

[5] Seidl Helmut. Haskell overloading is DEXPTIME-complete. *Information Processing Letters*, 52(2):57–60, 1994.

[6] Brian O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 2008.

[7] Alejandro Serrano Mena. *Beginning Haskell: A Project-Based Approach*. Apress, 2014.

[8] Simon L. Peyton Jones, Cordy Hall, Kevin Hammond, Jones Cordy, Hall Kevin, Will Partain, and Phil Wadler. The Glasgow Haskell Compiler: a technical overview. In *Proceedings of the UK Joint Framework for Information Technology (JFIT) Technical Conference*, volume 93, 1992.

[9] Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4):393–434, 2002.

[10] Simon L. Peyton Jones and Andre L. M. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, 1998.

[11] Simon Marlow and Simon Peytnon Jones. Multicore garbage collection with local heaps. In *Proceedings of the International Symposium on Memory Management*, pages 21–32, 2011.

[12] Rohit Jha, Alfy Samuel, Ashmee Pawar, and M. Kiruthika. A Domain-Specific Language for Discrete Mathematics. *International Journal of Computer Applications*, 70(15):6–19, 2013.

[13] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM (CACM)*, 21(2):120–126, 1978.

[14] John Kelsey, Bruce Schneier, and David Wagner. Mod n Cryptanalysis, with Applications against RC5P and M6. *Lecture Notes in Computer Science*, 1636(1999):139–155, 2001.

[15] Whitfield Diffie and Martin R. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[16] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[17] Wen-Jing Hsu. Fibonacci cubes: a new interconnection Topology. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):3–12, 1993.

[18] Bernard Kolman, Robert C. Busby, and Sharon Cutler Ross. *Discrete Mathematical Structures - Sixth Edition.* Pearson, 2008.

[19] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language - Second Edition.* Prentice-Hall, 1988.

[20] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[21] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen K. Lenstra, Emmanuel ThomÃľ, Joppe W. Bos, Pierrick Gaudry, Alexander Kruppa, Peter L. Montgomery, Dag A. Osvik, Herman T. Riele, Andrey Timofeev, and Paul Zimmermann. Factorization of a 768-bit RSA modulus. In *Proceedings of the 30th Annual Conference on Advances in Cryptology (CRYPTO 10)*, pages 333–350, 2010.