concept.cm

```
/*
    ==============================================================================

    Copyright (c) 2012-2015 Seppo Laakko
    http://sourceforge.net/projects/cmajor/

    Distributed under the GNU General Public License, version 3 (GPLv3).
    (See accompanying LICENSE.txt or http://www.gnu.org/licenses/gpl.html
        )

 ==============================================================================
    */

// Copyright (c) 1994
// Hewlett-Packard Company
// Copyright (c) 1996
// Silicon Graphics Computer Systems, Inc.
// Copyright (c) 2009 Alexander Stepanov and Paul McJones

using System;

namespace System.Concepts
{
    public concept DefaultConstructible<T>
    {
        T();
    }

    public concept CopyConstructible<T>
    {
        T(const T&);
        axiom copyIsEqual(T a) { eq(T(a), a); }
    }

    public concept MoveConstructible<T>
    {
        T(T&&);
    }

    public concept Destructible<T>
    {
        ~T();
    }

    public concept CopyAssignable<T>
    {
        void operator=(const T&);
    }

    public concept CopyAssignable<T, U>
    {
```

```
    void operator=(const U&);
}

public concept MoveAssignable<T>
{
    void operator=(T&&);
}

public concept Copyable<T>
{
    where T is CopyConstructible and T is CopyAssignable;
}

public concept Movable<T>
{
    where T is MoveConstructible and T is MoveAssignable;
}

public concept Semiregular<T>
{
    where T is DefaultConstructible and (T is Copyable or T is
        Movable) and T is Destructible;
}

public concept EqualityComparable<T>
{
    bool operator==(T, T);
    axiom equal(T a, T b) { a == b <=> eq(a, b); }
    axiom reflexive(T a) { a == a; }
    axiom symmetric(T a, T b) { a == b => b == a; }
    axiom transitive(T a, T b, T c) { a == b && b == c => a == c; }
    axiom notEqualTo(T a, T b) { a != b <=> !(a == b); }
}

public concept EqualityComparable<T, U>: Common<T, U>
{
    where T is EqualityComparable and U is EqualityComparable and
        CommonType is EqualityComparable;
}

public concept LessThanComparable<T>
{
    bool operator<(T, T);
    axiom irreflexive(T a) { !(a < a); }
    axiom antisymmetric(T a, T b) { a < b => !(b < a); }
    axiom transitive(T a, T b, T c) { a < b && b < c => a < c; }
    axiom total(T a, T b) { a < b || a == b || a > b; }
    axiom greaterThan(T a, T b) { a > b <=> b < a; }
    axiom greaterThanOrEqualTo(T a, T b) { a >= b <=> !(a < b); }
    axiom lessThanOrEqualTo(T a, T b) { a <= b <=> !(b < a); }
}

public concept LessThanComparable<T, U>: Common<T, U>
```

```
{
    where T is LessThanComparable and U is LessThanComparable and
        CommonType is LessThanComparable;
}

public concept Regular<T>: Semiregular<T>
{
    where T is EqualityComparable;
}

public concept TotallyOrdered<T>: Regular<T>
{
    where T is LessThanComparable;
}

public concept TotallyOrdered<T, U>: Common<T, U>
{
    where T is TotallyOrdered and U is TotallyOrdered and CommonType
        is TotallyOrdered;
}

public concept TrivialIterator<T> where T is Semiregular
{
    typename T.ValueType;
    where T.ValueType is Semiregular;
    typename T.ReferenceType;
    where T.ReferenceType is T.ValueType&;
    T.ReferenceType operator*();
    typename T.PointerType;
    where T.PointerType is T.ValueType*;
    T.PointerType operator->();
}

public concept OutputIterator<T>: TrivialIterator<T>
{
    T& operator++();
}

public concept InputIterator<T>: TrivialIterator<T>
{
    T& operator++();
    where T is Regular;
}

public concept ForwardIterator<T>: InputIterator<T>
{
    where T is OutputIterator;
}

public concept BidirectionalIterator<T>: ForwardIterator<T>
{
    T& operator--();
}
```

```
public concept RandomAccessIterator<T>: BidirectionalIterator<T>
{
    T.ReferenceType operator[](int index);
    T operator+(T, int);
    T operator+(int, T);
    T operator-(T, int);
    int operator-(T, T);
    where T is LessThanComparable;
}

public concept UnaryFunction<T> where T is Semiregular
{
    typename T.ArgumentType;
    typename T.ResultType;
    where T.ArgumentType is Semiregular;
    T.ResultType operator()(T.ArgumentType);
}

public concept BinaryFunction<T> where T is Semiregular
{
    typename T.FirstArgumentType;
    typename T.SecondArgumentType;
    typename T.ResultType;
    where T.FirstArgumentType is Semiregular and T.SecondArgumentType
        is Semiregular;
    T.ResultType operator()(T.FirstArgumentType, T.SecondArgumentType
        );
}

public concept UnaryPredicate<T>: UnaryFunction<T>
{
    where T.ResultType is bool;
}

public concept BinaryPredicate<T>: BinaryFunction<T>
{
    where T.ResultType is bool;
}

public concept Relation<T>: BinaryPredicate<T>
{
// types Domain, FirstArgumentType and SecondArgumentType are all
   same type:
    typename T.Domain;
    where Same<T.Domain, T.FirstArgumentType> and Same<T.
        SecondArgumentType, T.Domain>;
}

public concept Relation<T, U, V>: BinaryPredicate<T>
{
    where T.FirstArgumentType is U and T.SecondArgumentType is V;
}
```

```
public concept UnaryOperation<T>: UnaryFunction<T>
{
    where T.ResultType is T.ArgumentType;
}

public concept BinaryOperation<T>: BinaryFunction<T>
{
    where T.ResultType is T.FirstArgumentType;
}

public concept HashFunction<T, Key>: UnaryFunction<T>
{
    where T.ArgumentType is Key and T.ResultType is ulong;
}

public concept KeySelectionFunction<T, Key, Value>: UnaryFunction<T>
{
    where T.ArgumentType is Value and T.ResultType is Key;
}

public concept Container<T> where T is Semiregular
{
    typename T.ValueType;
    typename T.Iterator;
    typename T.ConstIterator;
    where T.Iterator is TrivialIterator and T.ConstIterator is
        TrivialIterator and T.ValueType is T.Iterator.ValueType;
    T.Iterator T.Begin();
    T.ConstIterator T.CBegin();
    T.Iterator T.End();
    T.ConstIterator T.CEnd();
    int T.Count();
    bool T.IsEmpty();
}

public concept BackInsertionSequence<T>: Container<T>
{
    void T.Add(T.ValueType);
}

public concept FrontInsertionSequence<T>: Container<T>
{
    T.Iterator T.InsertFront(T.ValueType);
}

public concept ForwardContainer<T>: Container<T>
{
    where T.Iterator is ForwardIterator and T.ConstIterator is
        ForwardIterator;
}

public concept InsertionSequence<T>: ForwardContainer<T>
```

```
{
    T.Iterator T.Insert(T.Iterator, T.ValueType);
}

public concept BidirectionalContainer<T>: ForwardContainer<T>
{
   where T.Iterator is BidirectionalIterator and T.ConstIterator is
      BidirectionalIterator;
}

public concept RandomAccessContainer<T>: BidirectionalContainer<T>
{
    where T.Iterator is RandomAccessIterator and T.ConstIterator is
       RandomAccessIterator;
}

public concept Integer<I> where I is TotallyOrdered
{
    I operator-(I);       // unary minus
    I operator~(I);       // complement
    I& operator++(I&);    // increment
    I& operator--(I&);    // decrement
    I operator+(I, I);    // addition
    I operator-(I, I);    // subtraction
    I operator*(I, I);    // multiplication
    I operator/(I, I);    // division
    I operator%(I, I);    // remainder
    I operator<<(I, I);   // shift left
    I operator>>(I, I);   // shift right
    I operator&(I, I);    // bitwise and
    I operator|(I, I);    // bitwise or
    I operator^(I, I);    // bitwise xor
}

public concept SignedInteger<I> where I is Integer
{
    I(sbyte);                    // implicit conversion from sbyte
       0..127
}

public concept UnsignedInteger<U> where U is Integer
{
    U(byte);                     // implicit conversion from byte
       0..255
}

public concept AdditiveSemigroup<T> where T is Regular
{
    T operator+(T, T);
    axiom additionIsAssociative(T a, T b, T c) { (a + b) + c == a + (
       b + c); }
    axiom additionIsCommutative(T a, T b) { a + b == b + a; }
}
```

```
public concept MultiplicativeSemigroup<T> where T is Regular
{
    T operator*(T, T);
    axiom multiplicationIsAssociative(T a, T b, T c) { (a * b) * c ==
        a * (b * c); }
}

public concept OrderedAdditiveSemigroup<T>: AdditiveSemigroup<T>
{
    where T is TotallyOrdered;
    axiom additionPreservesOrder(T a, T b, T c) { a < b => a + c < b
       + c; }
}

public concept OrderedMultiplicativeSemigroup<T>:
   MultiplicativeSemigroup<T>
{
    where T is TotallyOrdered;
}

public concept AdditiveMonoid<T>: AdditiveSemigroup<T>
{
    T(sbyte); // ensure zero can be converted to T
    axiom zeroIsIdentityElement(T a) { a + 0 == a && 0 + a == a; }
}

public concept MultiplicativeMonoid<T>: MultiplicativeSemigroup<T>
{
    T(sbyte); // ensure one can be converted to T
    axiom oneIsIdentityElement(T a) { a * 1 == a && 1 * a == a; }
}

public concept AdditiveGroup<T>: AdditiveMonoid<T>
{
    T operator-(T);
    axiom unaryMinusIsInverseOp(T a) { a + (-a) == 0 && (-a) + a ==
       0; }
    T operator-(T, T);
    axiom subtract(T a, T b) { a - b == a + (-b); }
}

public concept MultiplicativeGroup<T>: MultiplicativeMonoid<T>
{
    // 1/a is multiplicative inverse
    axiom multiplicativeInverseIsInverseOp(T a) { a * (1/a) == 1 &&
       (1/a) * a == 1; }
    T operator/(T, T);
    axiom division(T a, T b) { a / b == a * (1/b); }
}

public concept OrderedAdditiveMonoid<T>: OrderedAdditiveSemigroup<T>
{
```

```
        where T is AdditiveMonoid;
    }

    public concept OrderedAdditiveGroup<T>: OrderedAdditiveMonoid<T>
    {
        where T is AdditiveGroup;
    }

    public concept Semiring<T>: AdditiveMonoid<T> where T is
       MultiplicativeMonoid
    {
        axiom zeroIsNotOne { 0 != 1; }
        axiom multiplyingByZeroYieldsZero(T a) { 0 * a == 0 && a * 0 ==
            0; }
        axiom distributivity(T a, T b, T c) { a * (b + c) == a * b + a *
            c && (b + c) * a == b * a + c * a; }
    }

    public concept CommutativeSemiring<T>: Semiring<T>
    {
        axiom multiplicationIsCommutative(T a, T b) { a * b == b * a; }
    }

    public concept EuclideanSemiring<T>: CommutativeSemiring<T>
    {
        T operator%(T, T);
        T operator/(T, T);
        axiom quotientAndRemainder(T a, T b) { b != 0 => a == a / b * b +
             a % b; }
    }
}
```