Cmajor Concepts

Seppo Laakko

February 27, 2014

1 Introduction

In Cmajor programming language functions and classes can be parameterized with types. A set of syntactic and semantic requirements for the actual types that can correctly be substituted for a formal type parameter in such a parameterized function or class is called a *concept*.

Concepts as a feature of a programming language enable the compiler to give better error messages when a substituted type does not satisfy syntactic requirements. They also enable the compiler to check that a formal type parameter constrained by a concept has all the necessary operations when it compiles a body of a parameterized function.

In this tutorial we will look over the tools that Cmajor programming language has to offer with respect to concepts.

2 Signature Constraints

In Cmajor programming language functions and classes can be parameterized with types. A parameterized function is called a *function template* and a parameterized class is called a *class template*. Cmajor follows the same naming convention as C++ in this respect. Figure 1 shows an example of a function template in Cmajor.

Figure 1: Function Template

```
T Add<T>(T a, T b)
{
    return a + b;
}
```

In case of Add function template it is natural to ask, what the type parameter T can be. Surely one must be able to pass object of type T as a parameter and return it by value. This requires that one can copy objects of type T, i.e. T must be *copy constructible*. One must also be able to add together two objects of type T. For now we can say that T must be additive. In Cmajor those requirements for template parameter types can be collected together and form a *concept*. Figure 2 shows the definition of this additive concept.

The requirement that T must be copy constructible is expressed by `T(const T&)` and the requirement that one must be able to add two objects of T is expressed by `T operator+(T, T)`. A synonym for a syntactic requirement like the preceding is a *constraint*, because it constrains

Figure 2: Additive Concept

```
concept Additive<T>
{
    T(const T&);
    T operator+(T, T);
}
```

what type T can be. Requirements that one must be able to do certain operations with objects of type T are called *signature constraints* in Cmajor.

Now we can rewrite the Add function template with Additive concept requirements to form a *constrained function template*. Figure 3 shows the rewritten Add function template.

Figure 3: Constrained Function Template

```
T Add<T>(T a, T b) where T is Additive
{
    return a + b;
}
```

The **where** clause states that the template parameter T must satisfy the requirements of Additive concept.

Let us now see what the Cmajor compiler thinks about the preceding definitions. When compiling the test program in figure 4 the Add function call with two integer arguments compiles fine, because surely one can copy integers and add them together. However calling the Add function with two Boolean values fails with an error message like this:

```
Error T12 in file C:/Programming/cmajor++/doc/tutorial/concepts/examples/additiv
e/additive.cm at line 19:
overload resolution failed: 'Add(bool, bool)' not found. Candidates:
type 'bool' does not satisfy the requirements of concept 'Additive' because
there is no member function with signature 'bool bool.operator+(bool)' and no fu
nction with signature 'bool operator+(bool, bool)':
    bool sb = Add(true, false); //  error
                ^
```

It means that the Add function template has not been instantiated for **bool** type, because **bool** does not conform to Additive concept. It does not conform, because there is no `operator+` member function in Boolean type and no free `operator+` function that takes Boolean parameters. Because of these facts, the overload resolution fails with the `Add(bool, bool)` function not found.

3 Associated Types and Typename Constraints

Techniques used in generic programming are somewhat different to traditional object oriented programming. In generic programming the types often do not form hierarchies, but they are

Figure 4: Additive Test

```
using System;

concept Additive<T>
{
    T(const T&);
    T operator+(T, T);
}

T Add<T>(T a, T b) where T is Additive
{
    return a + b;
}

void main()
{
    int a = 1;
    int b = 2;
    int si = Add(a, b);                 // ok
    bool sb = Add(true, false);         // error
}
```

otherwise related: one type is *associated* with another. An associated type is defined inside another type or there is a **typedef** inside a type that names the associated type.

For example in List class template the type contained in list is an associated type of the List type, and it is named ValueType (Fig. 5.)

Figure 5: List Class Template

```
class List<T> where T is Semiregular
{
    public typedef T ValueType;
//  ...
}
```

To express that a type has an associated type in relation to Cmajor concepts, one can define a *typename constraint* in a concept definition. For example in the definition of Container concept there is a requirement that a type conforming to Container concept has an associated type called ValueType (Fig. 6.)

The List class template has the associated type ValueType in form of **typedef** definition, so it conforms to Container concept in this respect.

4 Embedded Constraints

A concept definition can state additional requirements for the template type parameter or for an associated type in form of an *embedded constraint*.

Figure 6: Container Concept

```
concept Container<T>
{
    typename ValueType;
//  ...
}
```

For example Semiregular concept (Fig. 7) has an embedded constraint that a semiregular type is DefaultConstructible, CopyConstructible, Destructible and Assignable, where DefaultConstructible, CopyConstructible, Destructible and Assignable are concepts defined elsewhere.

Figure 7: Semiregular Concept

```
concept Semiregular<T>
{
    where T is DefaultConstructible and T is CopyConstructible and T is
        Destructible and T is Assignable;
}
```

5 Multiparameter Constraints

Sometimes a concept involves two or more types. For example Assignable<T, U> (Fig. 8) concept states the requirement that type T implements an assignment operator that takes type U parameter.

Figure 8: Multiparameter Concept

```
concept Assignable<T, U>
{
    void operator=(const U&);
}
```

6 Concept Hierarchies

In generic programming classes do not often form hierarchies, while concepts often do. When a class derives from another class, it inherits the members of the base class and it can *override* virtual functions defined in the base class. With respect to concepts, a concept can *refine* another concept. When concept refines another, it can define additional requirements and *redefine* requirements defined in the refined concept for parameterized types and for types associated with the parameterized types.

6.1 Iterator Hierarchy

The Cmajor System library borrows and adapts the implementation of many algorithms, containers and concepts from the C++ Standard Template Library [5], among other things iterator concepts. Iterators in Cmajor form a refinement hierarchy shown in figure 9.

Figure 9: Iterator Hierarchy

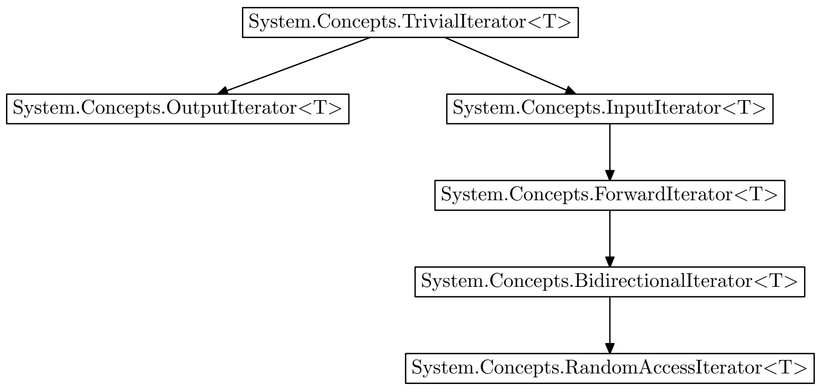
System.Concepts.TrivialIterator<T>
System.Concepts.OutputIterator<T>
System.Concepts.InputIterator<T>
System.Concepts.ForwardIterator<T>
System.Concepts.BidirectionalIterator<T>
System.Concepts.RandomAccessIterator<T>

Figure 10: Trivial, Input and Output Iterators

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

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

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

The TrivialIterator (Fig. 10) is a concept that collects the common typename and signature requirements for all iterator types. There is no class in Cmajor that is a model of bare trivial iterator concept. Every iterator must define associated types ValueType, Reference-Type and PointerType: these are typename constraints. ValueType must be semiregular:

this is an embedded constraint. Every iterator must also define `operator->` function that returns a PointerType: this is a signature constraint.

The OutputIterator concept refines the TrivialIterator concept by adding `operator++` signature constraint. We can say that an output iterator is a trivial iterator that can be incremented. Likewise an input iterator is a trivial iterator that can be incremented and compared for equality and inequality. That is the requirement added by the Regular concept.

Figure 11: Forward, Bidirectional and Random Access Iterators

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

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

concept RandomAccessIterator<T>: BidirectionalIterator<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;
}
```

A forward iterator (Fig. 11) refines in a sense both input and output iterators, but Cmajor lacks multiple refinement, so the requirement that forward iterator is an output iterator is expressed as an embedded constraint. Additionally, a forward iterator is a multipass input iterator, but this requirement cannot be expressed in Cmajor syntax.

Some algorithms require an iterator that can walk both forwards and backwards, but cannot “jump”: this is what a bidirectional iterator can do.

Finally some algorithms require pointer-like operations: indexing, moving arbitrary offsets forwards and backwards, computing the distance between two iterators and comparing two iterators with less than, greater than, less or equal to and greater or equal to relations. This is what a random access iterator can do.

In analogy to class inheritance where a derived class can be used in place of the base class (the *derived* IS-A *base*), the same applies to concept refinement: a type that conforms to a more constrained concept, can be used in context where a type that conforms only to a less constrained concept is required. Thus a random access iterator IS-A bidirectional iterator IS-A forward iterator.

6.2 Container Hierarchy

The container hierarchy goes parallel with the iterator hierarchy.

Figure 12: Container Concept

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

The container concept (Fig. 12) defines some common requirements for all container types. It requires that the container has associated iterator and constant iterator types that conform to trivial iterator concept.

It also links the value type of the container to the value type of the iterator by requiring them to be the same type. It finally requires that the container has member functions that return iterator types and some other member functions common to all containers.

Figure 13: Forward Container Concept

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

A forward container concept (Fig. 13) redefines the requirement in container concept for the iterator type to be not just a trivial iterator but a forward iterator. Likewise bidirectional container refines forward container and redefines the requirement for the iterator type to be not just a forward iterator but a bidirectional iterator, and random access container refines bidirectional container and redefines the iterator type to be not just a bidirectional iterator but a random access iterator.

7 Built-in Concepts

There are some concepts built into the Cmajor language.

The **Same<T, U>** concept sets a requirement that T and U are exactly the same type. For example if A is **const int&**, B is **const int&** and C is **int&**, Same<A, B> is true, but Same<A, C> is false.

The **Derived<T, U>** concept sets a requirement that T and U are class types and T is derived from U.

The **Convertible<T, U>** concept sets a requirement that type T is implicitly convertible to U. For example **int** is implicitly convertible to **double**, but not vice versa.

The **ExplicitlyConvertible<T, U>** concept sets a requirement that T can be implicitly or explicitly converted to U. For example **double** is explicitly convertible to **int** (by using a cast), but **void*** is not explicitly convertible to **int**.

The **Common<T, U>** concept sets a requirement that T and U have a common type that both are convertible to. The concept exposes the common type as a typedef CommonType. For example **Common<int, double>** is true and their common type is **double**, but **Common<void*, int>** is false.

8 Axioms

So far the requirements for parameterized types have been purely syntactic — the type must have certain operation or it must have an associated type of the given name.

In Cmajor one can also express semantic requirements for parameterized types in form of *axioms* (Fig. 14.)

Figure 14: Equality Comparable Concept

```
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); }
}
```

Axioms are not processed by the Cmajor compiler in any other way than parsing their syntax. Axioms are documentation for the programmer so the programmer can reason how a type that models a concept behaves.

9 Using Concepts

Now we have roughly walked through the tools Cmajor has to offer for defining concepts. So how do we use them?

9.1 Generic Algorithm

Let us take a look at the copy algorithm that copies an input sequence to an output sequence. The Copy function has a **where** clause that lists the requirements for the template parameters I and O combining them with **and** connective (Fig. 15.)

First Copy requires that I template parameter is a model of an input iterator, because we need it to iterate through the input sequence and to test whether we have reached the

Figure 15: Copy Algorithm

```
public O Copy<I, O>(I begin, I end, O to)
    where I is InputIterator and O is OutputIterator and
    Assignable<O.ValueType, I.ValueType>
{
    while (begin != end)
    {
        *to = *begin;
        ++begin;
        ++to;
    }
}
```

end of the input sequence. That is what input iterators can do. Then it requires that O template parameter is a model of an output iterator, because we need it to generate the output sequence. That is what output iterators can do. Finally we need to assign an object that is of type ValueType associated with input iterator type (I.ValueType [1]) to an object that is of type ValueType associated with output iterator type (O.ValueType). That is how we copy values from input sequence to output sequence. It takes a form of a multiparameter Assignable<T, U> constraint.

You can pass random access iterators to the Copy function (because they are input and output iterators), but the algorithm requires only input and output iterators. This makes it generic.

9.2 Container Concepts and Overloading

Figure 16: Sorting a Container

```
public void Sort<C>(C& c)
    where C is RandomAccessContainer and C.Iterator.ValueType is
        TotallyOrdered
{
    // ...
}

public void Sort<C>(C& c)
    where C is ForwardContainer and C.Iterator.ValueType is
        TotallyOrdered
{
    List<C.ValueType> list;
    Copy(c.CBegin(), c.CEnd(), BackInserter(list));
    Sort(list);
    Copy(list.CBegin(), list.CEnd(), c.Begin());
}
```

[1] Associated types are referred to using the **.** notation in Cmajor.

Let us take a look at the sort algorithm for containers (Fig. 16.) It has two overloads. Both take a container parameter, but one constrains the container to be a random access container, and the other constrains the container to be just a forward container.

Suppose you have a `List<int> list` that need to be sorted [2]. List is a model of a random access container, because the iterators it provides are random access iterators.

Now you call `Sort(list)`. The compiler checks the sort overloads. First it finds that sort overload for a random access container is a match, because List is a random access container. Then it finds that the sort overload for a forward container matches also, because the constraint check for a forward container succeeds, when the container is actually a random access container. However the sort for a random access container is a better match, because random access container concept refines forward container concept, so the sort for a random access container gets instantiated and called.

Suppose now that you have a `ForwardList<int> fwdlist`. System.ForwardList is a model of a forward container [3]. Now you call `Sort(fwdlist)`. Again the compiler checks the sort overloads, but this time it rejects the sort for a random access container, because ForwardList fails the constraint check for this overload. So the one that gets instantiated and called this time is the sort for a forward container — a slower version of the sort algorithm.

9.3 Iterator Concepts and Overloading

First take a look at overloads of Distance function that returns a distance between two iterators. It has two overloads.

Figure 17: Distance for Forward Iterators

```
public nothrow int Distance<I>(I first, I last)
    where I is ForwardItetor
{
    int distance = 0;
    while (first != last)
    {
        ++first;
        ++distance;
    }
    return distance;
}
```

Figure 18: Distance for Random Access Iterators

```
public nothrow inline int Distance<I>(I first, I last)
    where I is RandomAccessIterator
{
    return last - first;
}
```

[2]In spite of its name, System.List is functionally equal to STL's std::vector class template.
[3]System.ForwardList is a singly linked list.

The first one is a slow version for a forward iterator (Fig. 17.) It counts the number of steps it takes to reach iterator *last* from iterator *first*.

The second one is a fast version for a random access iterator (Fig. 18.) It simply returns the difference of *last* and *first* iterators. It can do this because random access iterators support this operation.

Figure 19: Next for Forward Iterators

```
public nothrow I Next<I>(I i, int n)
    where I is ForwardIterator
{
    #assert(n >= 0);
    while (n > 0)
    {
        ++i;
        --n;
    }
    return i;
}
```

Figure 20: Next for Random Access Iterators

```
public nothrow inline I Next<I>(I i, int n)
    where I is RandomAccessIterator
{
    return i + n;
}
```

Then look at the overloads of a Next function, that returns an iterator advanced the specified number of steps. Again two overloads, one for forward iterators and one for random access iterators. The first version (Fig. 19) increments a forward iterator given number of steps. The second version (Fig. 20) simply returns a sum of a random access iterator and an offset.

Figure 21: Lower Bound

```
public nothrow I LowerBound<I, T>(I first, I last, const T& value) where
    I is ForwardIterator and TotallyOrdered<T, I.ValueType>
{
    int len = Distance(first, last);
    while (len > 0)
    {
        int half = len >> 1;
        I middle = Next(first, half);
        if (value > *middle)
        {
            first = middle;
            ++first;
            len = len - half - 1;
        }
        else // value <= *middle
        {
            len = half;
        }
    }
    return first;
}
```

Finally take a look at the lower bound function (Fig. 21.) It searches a value in a sorted sequence using binary search and returns an iterator pointing to the first position that has a value that is equal to or greater than the given value. It uses the Distance and Next helper functions.

Now comes the clue of this story. There is only one version and that is for the forward iterator. Where's the random access version? We don't need it, because when the lower bound function is instantiated with a random access iterator, the compiler instantiates the fast random access versions of the Distance and Next functions and they will be called.

References

[1] A Concept Design for the STL
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3351.pdf

[2] Concepts: Linguistic Suppport for Generic Progamming in C++
http://www.osl.iu.edu/publications/prints/2006/Gregor06:Concepts.pdf

[3] Concepts Lite: Constraining Templates with Predicates
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3580.pdf

[4] Design of Concept Libraries for C++
http://www.stroustrup.com/sle2011-concepts.pdf

[5] Standard Template Library Programmer's Guide
http://www.sgi.com/tech/stl/