
# The benefits of concepts

PETER GOTTSCHLING – SMARTSOFT/TU DRESDEN

TECHNISCHE UNIVERSITÄT DRESDEN

SmartSoft – TU Dresden
Autor@smartsoft-computing.com
Tel.: +49 (0) 351 463 34018


## *Goals for Concepts*

- More concise and meaningful error messages
- Retroactive interface adaption
- Expressing semantic behavior
- Semantic verification (Bergen team)
  - Language support for full formal specification?
- Generic Optimization
- Compiler support (responsibility) for a higher quality of software development
  - Instead of external specification/verification tools

# Current Concept Support


## *More concise error message*

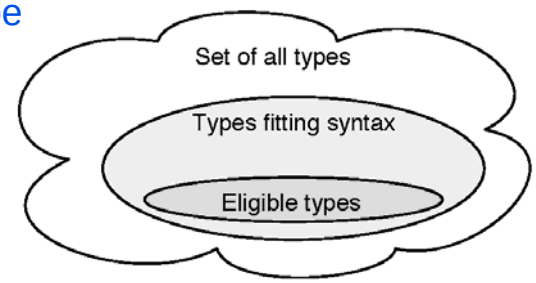
```
conceptg++  sort_and_concepts.cpp   -o sort_and_concepts
sort_and_concepts.cpp: In function 'int main(int, char**)':
sort_and_concepts.cpp:19: error: no matching function for call to
    'sort(__gnu_cxx::__normal_iterator<const int*, std::vector<int,
    std::allocator<int> > >, __gnu_cxx::__normal_iterator<const
    int*, std::vector<int, std::allocator<int> > >)'
/Users/pgottsch/software/conceptgcc-config-darwin/powerpc-
    apple-darwin7.3.1/libstdc++-v3/include/bits/stl_algo.h:2913:
    note: candidates are: void std::sort(_Iter, _Iter) [with _Iter =
    __gnu_cxx::__normal_iterator<const int*, std::vector<int,
    std::allocator<int> > >] <where clause>
sort_and_concepts.cpp:19: note:   no concept map for
    requirement
    'std::MutableRandomAccessIterator<__gnu_cxx::__normal_it
    erator<const int*, std::vector<int, std::allocator<int> > > >'
sort_and_concepts.cpp:19: note:   no concept map for
    requirement 'std::Assignable<const int&, const int&>'
```


## Express semantics

```
concept VectorSpace<typename Vector, typename Scalar = Vector::value_type>
  : AdditiveAbelianGroup<Vector>, Multiplicable<Scalar, Vector>,
    Multiplicable<Vector, Scalar>
VectorSpace<typename AddOp, typename MultOp, typename Element>
{
  requires Field<Scalar>;

  axiom Distributivity(Scalar a, Scalar b, Vector v, Vector w)
  {
    a * (v + w) == a * v + a * w; (v + w) * a == v * a + w * a;
    (a + b) * v == a * v + b * v; v * (a + b) == v * a + v * b;
  }
}

auto concept BanachSpace<typename N, typename Vector,
                         typename Scalar = Vector::value_type >
  : Norm<N, Vector, Scalar>, VectorSpace<Vector, Scalar>
{}

auto concept HilbertSpace<typename I, typename Vector,
                          typename Scalar = Vector::value_type >
  : InnerProduct<I, Vector, Scalar>,
    BanachSpace<induced_norm_t<I, Vector, Scalar>, Vector, Scalar>
{}
```


## Explicit vs. implicit concepts

- Compiler cannot recognize semantics
  - Models of semantic concepts must be declared by programmer
  - Not always distinguishable syntactically



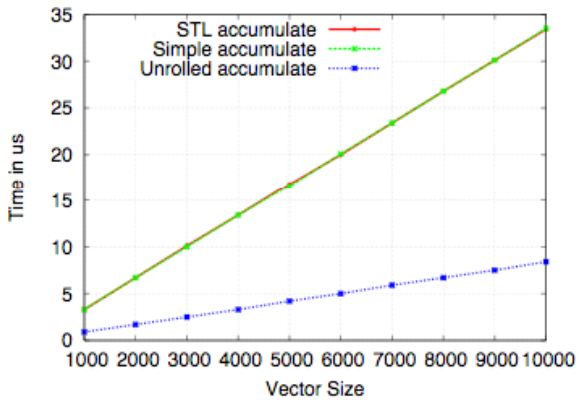

## Optimize on semantics

```
template <RandomAccessIterator Iter, typename T, typename Op>
    requires Monoid<Op, T> && Commutative<Op, T>
T accumulate(Iter first, Iter last, T s, Op op)
{
  T s1= identity(op, s), s2= s1, s3= s2;
  for (last_block= …;first != last_block; first+= 4) {
    s = op(s, *first); s1 = op(s1, *(first+1));
    s2 = op(s2, *(first+2)); s3 = op(s3, *(first+3)); }
  for (;first != last; ++first) s = op(s, *first);
  return op(op(s, s1), op(s2, s3));
}
```

- Or in compiler based on axioms (Jaakko/Jeremiah)


## Accelerated accumulate

- Addition of integer



SmartSoft

## *Challenges Anisomorphisms*

- Between mathematics and C++
- Math set can be multiple types
  - E.g. float, double
  - Infinite number with expression templates
- C++ type can be multiple sets
  - Vector for each set
- C++ functions are sets of mathematical functions
  - Due to overloading and templates

## DESIRED CONCEPT EXTENSIONS


## Dynamic concepts

- Are concepts on objects
- Motivation: for function arguments only property of object matters not of whole type, e.g.

```
template <typename LinearOperator, …>
   requires …
void conjugate_gradient(const LinearOperator& A, …)
   requires Symmetric(A) && PositiveDefinite(A)
{}
```


## Dynamic concepts (cont)

- Allow for (dynamic) overloading
- Opposed to assertion-based preconditions

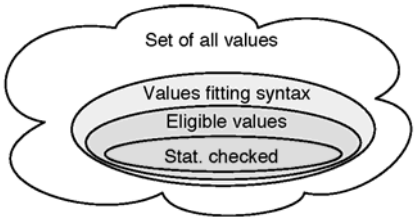
```
template <T>
   requires st1<T>
void f(const T& x)   {}

template <T>
   requires st1<T> && st2<T>
void f(const T& x)
{
   assert(dyn(x)); // crash
}
```

```
template <T>
   requires st1<T>
void f(const T& x)   {}

template <T>
   requires st1<T> && st2<T>
void f(const T& x)
   requires dyn(x) // use other
{}
```

SmartSoft

# Dynamic Concepts (cont)

- Are extensions of static concepts
- A type modeling a concept → all values/objects are dynamic models
- Can replace run-time by compile-time check

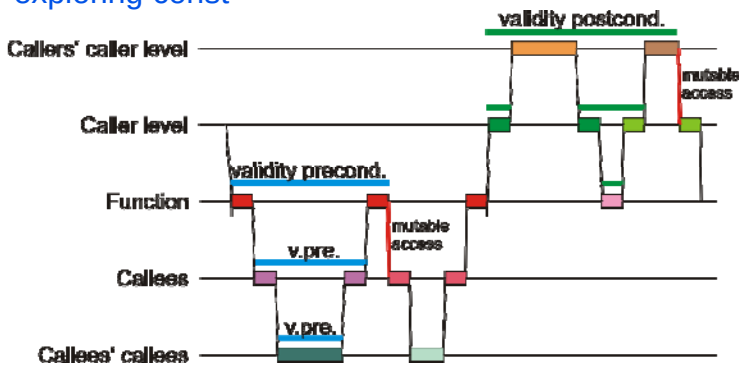



```
template <typename LinearOperator, …>
   requires …
void cg(const LinearOperator& A, …)
   requires Symmetric(A)
         && PositiveDefinite(A)
{}
```

SmartSoft

# Dynamic concepts (cont) Property conservation

- Allow for requiring pre- and declaring post-conditions
- Properties are valid as long as objects are not modified → exploring const




## *Dynamic concepts (cont) Increasing constancy*

- Axiom-based optimizations allowed within validity range of property
- Maximizing constancy in arguments extents validity
  - Result-dependent argument type ordering

```
template<class T> T f(const T& x);

matrix_type A;
const double x= A[i][j];
y= f(A[i][j]);
```

- We could use const version of operator[]
- For constant result prioritize constant arguments
- Increases reliability even without dynamic concepts


## *Inferring syntactic concepts from missing constraints*

- Syntactical constraints are already contained in source
- Often much longer than code itself
- Distract from semantic constraints
- Can be generated by the compiler
- Parser-based documentation tools needed

SmartSoft

## *Example for inferring syntactic concepts*

```
template <typename Op, std::Semiregular Element, Integral Exponent>
  requires Group<Op, Element>
  && std::Convertible<std::Callable2<Op, Element, Element>::result type, Element>
  && std::Semiregular<math::Inversion<Op, Element>::result type>
  && std::HasNegate<Exponent>
  && math::Monoid<Op, math::Inversion<Op, Element>::result type>
  && Integral< std::HasNegate<Exponent>::result type>
  && std::Callable2<Op, math::Inversion<Op, Element>::result type,
                        math::Inversion<Op, Element>::result type>
  && std::Convertible<std::Callable2<Op, math::Inversion<Op, Element>::result type,
                                          math::Inversion<Op, Element>::result type>::result type,
                                math::Inversion<Op, Element>::result type>
inline Element power(const Element& a, Exponent n, Op op)
{
   return n < 0 ? multiply and square(inverse(op, a), -n, op)
                : multiply and square(a, n, op);
}
```

SmartSoft

## *What is code complexity?*

```
int m,u,e;float g,s,f;char*_=")xx.xxx@xxx(rezneuM drahnreB\n";main(){for(
    ;e<1944;){u=s=f=0;do{g=s*s-f*f-2.1+.035*(m=e%81);f=2*s*f+e/81*.088-1.1;s
    =g;}while(++u<19&&g*s+f*f<4);putchar(_[++e>1863&&m<28?27-
    m:m>79?28:u]);}}
```

- If you can write this correctly you don‘t need:
  - Concepts
  - Specification/verificatin
  - Type systems
- Sometimes more is less


## *Open questions (to leave you puzzled)*

- What properties can we prove?
  - Which ones do we want to?
  - Interface to Coq or alike?
- Do we want/need?
  - Easier renaming/aliasing → Magne
  - Transitivity of axioms (categories) → Marcin
  - template <typename T, concept C> class c;
  - Concepts of concepts, e.g. UnaryConcept
- Which feedback what the compiler did?