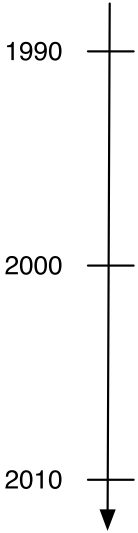
The C++0x Concept Effort

Jeremy Siek

University of Colorado at Boulder

SSGIP 2010

Bird's Eye View



Generic Programming hits the mainstream with the Standard Template Library in C++.

Generic Programming applied extensively in C++ libraries (Boost, etc.).

Scalability problems with C++ templates.

Researchers at Indiana and Texas start designing constrained templates, that is, "concepts".

"Concepts" voted into C++0x.

"Concepts" voted back out of C++0x.

Generic Programming

Generic Programming is
higher-order, polymorphic programming
scaled up to large, complex families of algorithms

Higher-Order, Polymorphic, circa 1970s

```
fun foldl f y nil = y
| foldl f y (x::xs) = foldl f (f (x,y)) xs
```

Typical uses:

```
foldl op + 1.0 [2.0,3.0,4.0];
val it = 10.0 : real

foldl op * 1 [2,3,4];
val it = 24 : int

foldl op @ [] [[1,2],[3],[4,5,6]];
val it = [4,5,6,3,1,2] : int list
```

Parametric Polymorphism: System F

Girard [1972], Reynolds [1974]

$$\frac{\Gamma, \alpha \vdash e : T}{\Gamma \vdash \Lambda\alpha.\ e : \forall\alpha. T}$$

$$\frac{\Gamma \vdash e : \forall\alpha. T}{\Gamma \vdash e[S] : [\alpha \mapsto S]T}$$

Scaling up to data structure genericity

```
fun foldl f y xs is_null head tail =
  if is_null xs then y
  else foldl f (f (head xs, y)) (tail xs) is_null head tail

fun array_is_null (a,i : int) = (i = Array.length a)
fun array_head (a,i) = Array.sub(a,i)
fun array_tail (a,i) = (a, i + 1)

val a = Array.fromList [2,3,4]

foldl op * 1 (a,0) array_is_null array_head array_tail

(* Too many arguments! *)
```

Generic Programming

"That is the fundamental point: algorithms are defined on algebraic structures."
— Alexander Stepanov

Algebraic Structures

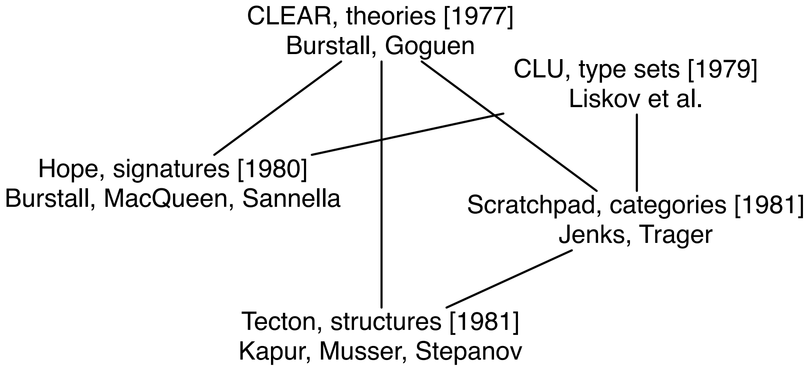
In Tecton [Kapur, Musser, Stepanov 1981]

```
create semigroup(S : set, +: S×S → S)
  with x + (y + z) = (x + y) + z;

create monoid(S : semigroup, 0: () → S)
  with 0 + x = x + 0 = x;

integers(I : set, + : I × I → I, ∗ : I × I → I,
         0 : () → I, 1 : () → I)

instantiate monoid of integers (S=I, + = +, 0 = 0)
instantiate monoid of integers (S=I, + = ∗, 0 = 1)
```

Algebraic Structures

```
create sequence(S : set, E : set, is_null : S → bool,
                head : S → E, tail : S → S);

provide sequence of monoid with
  reduction:
   x → if is_null(x) then 0
        else head(x) + reduction(tail(x))
```

Early Languages with Algebraic Structures



Generic Library Development in the 1980s

- Stepanov, Musser, and several colleagues
- Algorithms on sequences, e.g., sorting and searching
- Implementations of libraries in Scheme and Ada.
- Stroustrup added Templates to C++ in 1988.
- Work begins on what becomes the C++ Standard Template Library.

Modern C++ Lingo: Concepts and Models

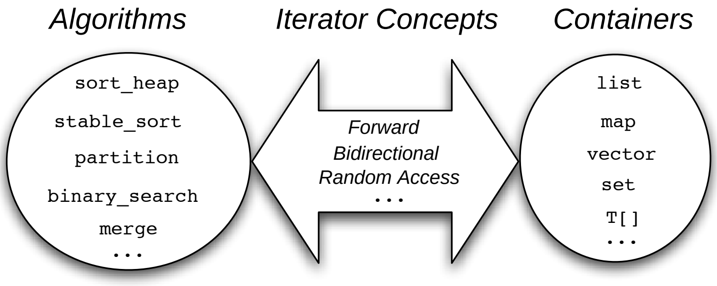
concept ≈ algebraic structure

model ≈ instance

Example

- Monoid is a concept.
- Sequence is a concept.
- int models Monoid.
- array<int> models Sequence.

Organization of the STL



The STL contains 112 generic algorithms and 12 container classes.

The Iterator Hierarchy

$$\begin{tikzcd}
 & & & \textbf{\textit{Input}} \\
\textbf{\textit{Random Access}} \arrow[r] & \textbf{\textit{Bidirectional}} \arrow[r] & \textbf{\textit{Forward}} \arrow[ur] \arrow[dr] & \\
 & & & \textbf{\textit{Output}}
\end{tikzcd}$$

- Different algorithms have different requirements.
- Concepts are created to serve in the specification of algorithms.
- Different perspective from traditional object-oriented approaches to class hierarchy design.

Example: Merge Algorithm from the STL

```
template <class InIter1, class InIter2, class OutIter>
OutIter
merge(InIter1 first1, InIter1 last1, InIter2 first2, InIter2 last2,
      OutIter result)
{
  while (first1 != last1 && first2 != last2) {
    if (*first2 < *first1) {
      *result = *first2; ++first2;
    } else {
      *result = *first1; ++first1;
    }
    ++result;
  }
  return copy(first2, last2, copy(first1, last1, result));
}
```

Type Requirements for Merge

```
template <class InIter1, class InIter2, class OutIter>
OutIter
merge(InIter1 first1, InIter1 last1, InIter2 first2, InIter2 last2,
      OutIter result);
```

Documentation:

- InIter1 models Input Iterator.
- InIter2 models Input Iterator.
- OutIter and the value type of InIter1 model Output Iterator.
- The value type of InIter1 models Less Than Comparable.
- The value type of InIter1 and InIter2 are the same type.

The Input Iterator Concept

Associated Types

- value type
- difference type

Valid Expressions

(X is the iterator type, T is the value type)

expression	**return type**	**semantics**
$*i$	Convertible to T	Returns the value at position `i`
$++i$	X&	Moves the iterator to the next position.
$i == j$	**bool**	Returns true if i and j are at the same position. $i == j$ implies $*i == *j$
$i != j$	**bool**	Equivalent to $!(i == j)$.

Complexity guarantees

All operations are amortized constant time.

The Semantics of C++ Templates

```
namespace std {
  template <class T>
  T min(T a, T b) {
    if (b < a) return b; else return a;
  }
}
int main() {
  return min(3, 4);
}
```

The Semantics of C++ Templates

```
namespace std {
  template <class T>
  T min(T a, T b) {
    if (b < a) return b; else return a;
  }
}
int main() {
  return min<int>(3, 4);
}
```

The Semantics of C++ Templates

```
namespace std {
  template <class T>
  T min(T a, T b) {
    if (b < a) return b; else return a;
  }
}
namespace std {
  template<>
  int min<int>(int a, int b) {
    if (b < a) return b; else return a;
  }
}
int main() {
  return min<int>(3, 4);
}
```

Type Checking Templates

Templates $\neq$ Parametric Polymorphism

```
// This would be an error if it were parametric
template <class T>
T min(T a, T b) {
  if (b < a) return b; else return a;
}
```

- At the point of a template's definition, the type checker may not check an expression whose type depends on a template parameter.
- The type checker doesn't have to check the other expressions, but it can.

Type Checking Templates

Template instantiations are type checked.

```
1  namespace std {
2    template <class T>
3    T min(T a, T b) { if (b < a) return b; else return a; }
4  }
5  struct A {};
6  int main() {
7    A a;
8    std::min(a, a);
9  }
```

The error message is:

```
error1.cpp: In function 'T std::min(T, T) [with T = A]':
error1.cpp:8: instantiated from here
error1.cpp:3: error: no match for 'operator<' in 'b < a'
```

Type Checking Templates: Types Variables

- Because templates are not fully checked until after instantiation, the type checker never has to deal with type variables.
- In contrast, with System F, the type checker does deal with type variables, and considers a variable equal only to itself.

The Infamous Error Messages

A simple misuse of a function template:

```
int main() {
  list<int> l;
  stable_sort(l.begin(), l.end());
}
```

```
stl_algo.h: In function 'void std::__inplace_stable_sort(_RandomAccessIter, _RandomAccessIter)
    [with _RandomAccessIter = std::_List_iterator<int, int&, int*>]':
stl_algo.h:2565: instantiated from 'void std::stable_sort(_RandomAccessIter, _RandomAccessIter)
    [with _RandomAccessIter = std::_List_iterator<int, int&, int*>]'
stable_sort_error.cpp:5: instantiated from here
stl_algo.h:2345: error: no match for 'std::_List_iterator<int, int&, int*>& -std::_List_iterator<int, int&, int*>&' operato
stl_algo.h:2565: instantiated from 'void std::stable_sort(_RandomAccessIter, _RandomAccessIter)
    [with _RandomAccessIter = std::_List_iterator<int, int&, int*>]'
stable_sort_error.cpp:5: instantiated from here
stl_algo.h:2349: error: no match for 'std::_List_iterator<int, int&, int*>& -std::_List_iterator<int, int&, int*>&' operato
stl_algo.h:2352: error: no match for 'std::_List_iterator<int, int&, int*>& -std::_List_iterator<int, int&, int*>&' operato
stl_algo.h:2352: error: no match for 'std::_List_iterator<int, int&, int*>& -std::_List_iterator<int, int&, int*>&' operato
stl_algo.h:2565: instantiated from 'void std::stable_sort(_RandomAccessIter, _RandomAccessIter)
    [with _RandomAccessIter = std::_List_iterator<int, int&, int*>]'
stable_sort_error.cpp:5: instantiated from here
stl_algo.h:2095: error: no match for 'std::_List_iterator<int, int&, int*>& + int' operator
stl_algo.h:2346: instantiated from 'void std::__inplace_stable_sort(_RandomAccessIter, _RandomAccessIter)
    [with _RandomAccessIter = std::_List_iterator<int, int&, int*>]'
stl_algo.h:2565: instantiated from 'void std::stable_sort(_RandomAccessIter, _RandomAccessIter)
    [with _RandomAccessIter = std::_List_iterator<int, int&, int*>]'
...
```

Concept Checking Tricks [Siek 2000]

```
template <class Iter>
struct RandomAccessIterator_concept {
  void constraints() { i += n; ... }
  iterator_traits<Iter>::difference_type n;
  Iter i;
};

template <class RandomAccessIter>
void stable_sort(RandomAccessIter first, RandomAccessIter last) {
  REQUIRE(RandomAccessIter, RandomAccessIterator);
  ...
}

#define REQUIRE(T, C) \
do { \
  void (C##_concept <T>::*x)() = \
    C##_concept <T>::constraints; \
  x = x; } while (0)
```

Template Definition Checking: Archetypes

```
template <class Base = null_archetype<> >
struct eq_archetype : public Base {
  // intentionally empty
};
template <class Base>
bool operator==(eq_archetype<Base>, eq_archetype<Base>)
  { return true; }
template <class Base>
bool operator!=(eq_archetype<Base>, eq_archetype<Base>)
  { return true; }

// fwd_iter_archetype ...

int main() {
  fwd_iter_archetype< eq_archetype<> > fo;
  fo = std::adjacent_find(fo, fo);
}
```

Commentary on Concept Checking Tricks

It works, but the code is boring to write and error prone.

Back to the Min Example, but with a UDT

```
namespace std {
  template <class T>
  T min(T a, T b) {
    if (b < a) return b; else return a;
  }
}
namespace L {
  class interval { };
  bool operator<(interval, interval);
}
int main() {
  L::interval i, j, k;
  k = std::min(i,j);
}
```

How does less-than in std::min resolve to L::**operator**<?

Argument Dependent Name Lookup (ADL)

```
namespace L {
  class interval { };
  bool operator<(interval, interval);
}
namespace std { // compiler generated instantiation of min
  template<>
  L::interval min<L::interval>(L::interval a, L::interval b) {
    if (b < a) return b; else return a;
  }
}
int main() {
  L::interval i, j, k;
  k = std::min<L::interval>(i,j);
}
```

ADL is Not Modular

```
namespace lib {
  template <class T> void load(T x, string)
    { printf("Proceeding as normal!\n"); }
  template<class T> void generic_fun(T x)
    { load(x, "file"); }
}
namespace N {
  struct b { int n; };
  template <class T> void load(T x, const char*)
    { printf("Hijacked!\n"); }
}
int main() {
  N::b a;
  lib::generic_fun(a);
}
// Output: Hijacked!
```

The Workaround: Qualify Everything (ugh!)

```
namespace lib {
  template <class T> void load(T x, string)
    { printf("Proceeding as normal.\n"); }
  template<class T> void generic_fun(T x)
    { lib::load(x, "file"); }
}
namespace N {
  struct b { int n; };
  template <class T> void load(T x, const char*)
    { printf("Hijacked!\n"); }
}
int main() {
  N::b a;
  lib::generic_fun(a);
}
// Output: Proceeding as normal.
```

Another Algorithm from the STL

```
template <class InIter, class T>
iterator_traits<InIter>::difference_type
count(InIter first, InIter last, T value) {
  iterator_traits<Iter>::difference_type n = 0;
  for ( ; first != last; ++first)
    if (*first == value)
      ++n;
  return n;
}
```

- The InputIterator concept requires an associated **difference type**.
- The iterator_traits class provides a means to map from an iterator to it's difference type.

Access to Associated Types, Take 1

The obvious approach is to just require the associated type as a nested typedef.

```
template <class T>
class list_iterator {
  typedef T value_type;
  typedef int difference_type;
  ...
};

template <class InIter, class T>
InIter::difference_type
count(InIter first, InIter last, const T& value);
```

But what if the iterator is just a pointer? (like **int**$*$?) Pointers can't have nested typedefs.

Associated Types via Traits [Myers 1995]

The solution relies on template specialization.

```
// the primary template
template <class T>
struct iterator_traits {
  typedef T::value_type value_type;
  typedef T::difference_type difference_type;
};
// partial specialization for pointers
template <class T>
struct iterator_traits<T*> {
  typedef T value_type;
  typedef std::ptrdiff_t difference_type;
};
int main() {
  iterator_traits<int*>::value_type x; // x : int
}
```

External Adaptation

A particularly nice thing about traits is that you can retroactively adapt third-party types.

```
namespace old_third_party {
  class old_iter { ... };
}

// adapt old_iter to InputIterator concept
namespace std {
  template<> struct iterator_traits<old_iter> {
    typedef old_iter::elt value_type; ...
  };
}
namespace old_third_party {
  old_iter::elt operator*(old_iter i) { return i.getCurrent(); }
  old_iter& operator++(old_iter i) { i.next(); return i; }
}
```

Yet Another Algorithm from the STL

```
template <class InIter, class Dist>
void advance_aux(InIter& i, Dist n, input_iterator_tag)
  { while (n--) ++i; }

template <class RandIter, class Dist>
void advance_aux(RandIter& i, Dist n, random_access_iterator_tag)
  { i += n; }

template <class InIter, class Dist>
void advance(InIter& i, Dist n) {
  typename iterator_traits<InIter>::iterator_category cat;
  advance_aux(i, n, cat);
}
```

- There's a tension between performance and generality.
- Tag dispatching resolves that tension by enabling selective dispatching based on iterator capabilities.

An Evaluation of C++ Templates

+ The instantiation-based compilation model induces little or no run-time overhead.
+ Invoking generic algorithms is convenient thanks to template argument deduction and ADL.
- ADL is not modular, so name resolution can go catastrophically wrong.
- Type checking templates is not modular, which hurts both the library implementor and the library user.
- Compilation time is a function of all template libraries in use (transitively). Thus, compile times can grow very large.

2003: Setting Out to Fix C++ Templates

- At Indiana University
  1. Ronald Garcia (Ph.D. student)
  2. Doug Gregor (Post doc)
  3. Jaakko Jarvi (Post doc)
  4. Andrew Lumsdaine (The Professor)
  5. Jeremy Siek (Ph.D. student)
  6. Jeremiah Willcock (Ph.D. student)
- At Texas A&M
  1. Bjarne Stroustrup (Creator of C++)
  2. Gabriel Dos Reis (Assistant Prof.)

Main Goal: Modular Type Checking

The primary goal is to provide the following guarantee: if the definition of a template type checks, then any instantiation that satisfies the templates type requirements will result in well typed code.

```
// template definition
template <class T> // require T to be Less Than Comparable
T min(T a, T b) {
  if (b < a) return b; else return a;
}

int main() {
  // template instantiation
  return min<int>(3, 4);
}
```

A Comparative Study [Indiana 2003]

Criterion	Definition
Multi-type concepts	Concepts can be collaborations of many types.
Multiple constraints	More than one constraint on a type parameter.
Associated type access	Mappings between types.
Constraints on assoc.	Concepts may constrain associated types.
Retroactive modeling	Modeling can be separate from type definition.
Separate compilation	Generic functions are independently compiled.
Implicit arg. deduction	Type arguments of a generic function can be deduced. Also, the language finds models to satisfy constraints of a generic function.
Modular type checking	Generic functions are independently checked.
Lexically scoped models	Model declarations are lexically scoped. Models may be imported from other namespaces.
Concept-based overloading	There can be multiple generic functions with the same name but differing constraints.
First-class functions	Anonymous functions with lexical scoping that are treated as first-class.

A Comparative Study [Indiana 2003]

	C++	SML	OCaml	Haskell	Java	C#
Multi-type concepts	-	●	○	●*	○	○
Multiple constraints	-	◒	◒	●	●	●
Associated type access	●	●	◒	◒	◒	◒
Constraints on assoc. types	-	●	●	◒	◒	◒
Retroactive modeling	-	●	●	●	○	○
Separate compilation	○	●	●	●	●	●
Implicit arg. deduction	●	○	●	●	●	●
Modular type checking	○	●	◒	●	●	●
Lexically scoped models	○	●	○	○	○	○
Concept-based overloading	●	○	○	○	○	○
Same-type constraints	-	●	○	◒	○	○
First-class functions	○	●	●	●	○	◒

*Using the multi-parameter type class extension to Haskell.

The Texas Proposal, at Kona Meeting 2003

Usage Patterns

```
concept Add {
  constraints(Add x) { Add y = x; x = x+y; }
};
```

Concept Composition

```
template <(C1 && C2) T> class X { ... };
template <(C1 || C2) T> class Y { ... };
template <(C1 && !C2) T> class Z { ... };
```

Concept Parameters (instead of associated types)

```
template <Value_type V> concept Forward_iterator {
  constraints(Forward_iterator p) {
    Forward_iterator q = p; V v = *p; p++; ++p;
  }
}
```

The Texas Proposal, at Kona Meeting 2003

Concept-based Overloading

```
template <InputIterator InIter>
void advance(InIter& i, InIter::difference_type n)
  { while (n--) ++i; }

template <RandomAccessIterator RandIter>
void advance(RandIter& i, RandIter::difference_type n)
  { i += n; }
```

The Texas Proposal, at Kona Meeting 2003

Implicit Modeling

```
class A { };
A operator+(A x, A y) { ... }

template<Add T>
void foo(T x) { ... }

int main() {
  A a;
  foo(a); // compiler deduces that A models Add
}
```

Issues with the Texas Proposal

```
concept Add {
  constraints(Add x) {                  Add::Add(Add);
    Add y = x; x = x+y;    ⟹           T1 operator+(Add, Add);
  }                                     Add::Add(T1);
};
```

Mapping from usage patterns to type signatures is complicated. It provides nice flexibility for the modeling type, but difficult to work with inside a function template. E.g.,

```
template<Add T> void g(T x, T y);

template<Add T> void f(T x, T y) {
  g(x, x + y); // error: no matching function for call to 'g(T, T1)'
}
```

Issues with the Texas Proposal, cont'd

- Open ended constraint language means compiler must be able to backtrack from arbitrarily deep instantiations. Poses an implementation difficulty.
- The || operator (or) on constraints causes type checking to become exponential. It was unclear what negation operator meant. Neither is used in any known library.
- Use of concept name as a type is confusingly similar to base classes.
- Implicit modeling and concept-based overloading interact badly.

Implicit Modeling and Overloading Don't Mix

```
template <class T>
class vector {
  template <InputIterator InIter>
  vector(InIter first, InIter last);

  template <ForwardIterator InIter>
  vector(InIter first, InIter last);
};

int main() {
  istream_iterator<int> i(cin), j;
  vector<int> v(i, j);
}
// Silently dispatches to the wrong constructor!
```

Research at Indiana

- Meanwhile, Siek, Jarvi, and Willcock are designing a prototype.
- We soon realize there is a tension between separate compilation and concept-based overloading.
- Even worse, there's tension between modular type checking and concept-based overloading.

Separate Compilation vs. Overloading

```
template <InputIterator Iter1, OutputIterator Iter2>
Iter2 copy(Iter1 first, Iter1 last, Iter2 result);

template <RandomAccessIterator Iter1, OutputIterator Iter2>
Iter2 copy(Iter1 first, Iter1 last, Iter2 result);

template <InputIterator InIter1, InputIterator InIter2,
          OutputIterator OutIter>
OutIter merge(InIter1 first1, InIter1 last1, InIter2 first2, InIter2 last2,
              OutIter result)
{ ...
  return copy(first2, last2, copy(first1, last1, result));
}
```

We can't resolve the calls to copy until we know the type of InIter1 and InIter2.

Modular Type Checking vs. Overloading

```
concept A { ... };
concept B : A { ... };

template<A T1, B T2> void f(T1, T2);
template<B T1, A T2> void f(T1, T2);

template<A T1, A T2> void g(T1 x, T2 y) { f(x, y); }

class X {};
models B<X> { };

int main() { X x; g(x, x); } // error, ambiguous
```

Research at Indiana

- We can't agree between separate compilation and two-phase concept-based overloading.
- So we investigate both.
- Siek develops a calculus with separate compilation, motivated by bad experiences with long compile times for Boost libraries.
- Jarvi and Willcock investigate two-phase concept-based overloading. C++ programmers want performance!

The Essential Bits: $\mathrm{F}^{\mathcal{G}}$ [Siek, 2005]

$$
\begin{array}{lcll}
c & \in & \text{Concept Names} & \\
\alpha, \beta & \in & \text{Type Variables} & \\
x, y & \in & \text{Term Variables} & \\
C & ::= & c\text{<}\overline{T}\text{>} \mid S = T & \text{Constraints} \\
S, T & ::= & \alpha \mid T \to T \mid \forall \alpha. T \mid C \Rightarrow T \mid \pi.\alpha & \\
e & ::= & x \mid e\ e \mid \lambda y : T.\ e \mid \Lambda \alpha.\ e \mid e[T] & \\
 & \mid & C \Rightarrow e \mid \textbf{type}\ \alpha = T\ \textbf{in}\ e & \\
 & \mid & \textbf{concept}\ c\text{<}\overline{\alpha}\text{>}\{\textbf{types}\ \overline{\beta};\ \textbf{requires}\ \overline{C};\ \overline{x : T}\}\ \textbf{in}\ e & \\
 & \mid & \textbf{model}\ c\text{<}\overline{T}\text{>}\ \{\ \textbf{types}\ \overline{\beta = S};\ \overline{x = e}\}\ \textbf{in}\ e & \\
 & \mid & \pi.x & \\
\pi & ::= & c\text{<}\overline{T}\text{>} \mid \pi.c\text{<}\overline{T}\text{>} & \\
\Gamma & ::= & \Gamma, x : T \mid \Gamma, \alpha \mid \Gamma, S = T \mid \Gamma, c\text{<}\overline{T}\text{>} \mid \Gamma, c\text{<}\overline{T}\text{>}\{\overline{\beta = S}\} &
\end{array}
$$

(This is inspired by Qualified Types, by Mark Jones.)

Example Concepts and Models in $\mathrm{F}^{\mathcal{G}}$

```
concept Semigroup<t> {
  binary_op : t→t→t;
} in
concept Monoid<t> {
  requires Semigroup<t>;
  identity_elt : t;
} in
model Semigroup<int> {
  binary_op = iadd;
} in
model Monoid<int> {
  identity_elt = 0;
} in
...
```

Generic Algorithm in $\mathrm{F}^{\mathcal{G}}$

```
let foldl =
  (Λ s. Seq<s> ⇒
    type t = Seq<s>.value_type in Monoid<t> ⇒
    fix (λ r : s → t. λ ls : s.
          let binary_op = Monoid<t>.binary_op in
          let identity_elt = Monoid<t>.identity_elt in
          if Seq<s>.is_null ls then identity_elt
          else binary_op(Seq<s>.head ls, r(Seq<s>.tail ls))))
in
   foldl[int list] [2,3,4]
```

Type Equality in $\mathrm{F}^{\mathcal{G}}$

$$\boxed{\Gamma \vdash S = T}$$

$$\frac{S = T \in \Gamma}{\Gamma \vdash S = T} \qquad \frac{\Gamma \vdash \pi = \pi'}{\Gamma \vdash \pi.\alpha = \pi'.\alpha}$$

$$\frac{\Gamma(\pi) = \{\overline{\beta = S}\}}{\Gamma \vdash \pi.\beta_i = S_i} \qquad \frac{\Gamma \vdash C_1 = C_2 \qquad \Gamma \vdash S = T}{\Gamma \vdash (C_1 \Rightarrow S) = (C_2 \Rightarrow T)}$$

$$\frac{\Gamma \vdash S_1 = T_1 \qquad \Gamma \vdash S_2 = T_2}{\Gamma \vdash S_1 \to S_2 = T_1 \to T_2} \qquad \frac{\Gamma \vdash S = [\beta \mapsto \alpha]T}{\Gamma \vdash \forall \alpha.S = \forall \beta.T}$$

and refl., symm., trans.

$\boxed{\Gamma \vdash C = C}$ $\cdots$

$\boxed{\Gamma \vdash \pi = \pi}$ $\cdots$

(This is inspired by type sharing in Standard ML.)

The Type System of $\mathrm{F}^{\mathcal{G}}$

$$\boxed{\Gamma \vdash e : T}$$

$$\frac{\Gamma, \textbf{concept}\ c{<}\alpha{>}\{\textbf{types}\ \overline{\beta};\ \textbf{requires}\ \overline{C};\ \overline{x : S}\} \vdash e : T}{\Gamma \vdash \textbf{concept}\ c{<}\alpha{>}\{\textbf{types}\ \overline{\beta};\ \textbf{requires}\ \overline{C};\ \overline{x : S}\}\ \textbf{in}\ e : T}$$

$$\frac{\begin{array}{c}\textbf{concept}\ c{<}\alpha{>}\{\textbf{types}\ \overline{\beta};\ \textbf{requires}\ \overline{C};\ \overline{x : T_2}\} \in \Gamma \\ \Gamma, \overline{\alpha = T_1}, \overline{\beta = T_3} \vdash \overline{C} \qquad \Gamma, \overline{\beta = T_3} \vdash \overline{e} : [\overline{\alpha \mapsto T_1}]\overline{T_2} \\ \Gamma, c{<}\overline{T_1}{>}\{\overline{\beta = T_3}\} \vdash e : T_4\end{array}}{\Gamma \vdash \textbf{model}\ c{<}\overline{T_1}{>}\ \{\ \textbf{types}\ \overline{\beta = T_3};\ \overline{x = e}\}\ \textbf{in}\ e : T_4}$$

The Type System of $\mathrm{F}^{\mathcal{G}}$, cont'd

$$\boxed{\Gamma \vdash e : T}$$

$$\frac{\Gamma \vdash e : C \Rightarrow T \qquad \Gamma \vdash C}{\Gamma \vdash e : T} \qquad \frac{\Gamma, C \vdash e : T}{\Gamma \vdash C \Rightarrow e : T}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : S \rightarrow T \\ \Gamma \vdash e_2 : S' \qquad \Gamma \vdash S = S'\end{array}}{\Gamma \vdash e_1 e_2 : T} \qquad \frac{\Gamma, \alpha = S \vdash e : T}{\Gamma \vdash \textbf{type}\ \alpha = S\ \textbf{in}\ e : T}$$

$$\frac{\textbf{concept}\ c{<}\alpha{>}\{\textbf{types}\ \overline{\beta};\ \textbf{requires}\ \overline{C};\ \overline{x : T}\} \in \Gamma \qquad \Gamma \vdash c{<}\overline{S}{>}}{\Gamma \vdash c{<}\overline{S}{>}.x_i : [\overline{\alpha \mapsto S}, \overline{\beta \mapsto c{<}\overline{S}{>}.\beta}]\, T_i}$$

$$\boxed{\Gamma \vdash C}$$

$$\frac{c{<}\overline{T'}{>} \in \Gamma \qquad \Gamma \vdash \overline{T = T'}}{\Gamma \vdash c{<}\overline{T}{>}} \qquad \frac{\Gamma \vdash T = T'}{\Gamma \vdash (T = T')}$$

Summary of $\mathrm{F}^{\mathcal{G}}$

- Modeling is explicit (via the **models** expression).
- $\mathrm{F}^{\mathcal{G}}$ has a modular type system. As in System F, a polymorphic expression is type checked independently of any use of the polymorphic expression. Similarly, a constrained expression is type checked independently of any use of the constrained expression.
- Type equality is implemented efficiently using algorithms for congruence closure (like SML).
- The dynamic semantics of $\mathrm{F}^{\mathcal{G}}$ is specified by a translation to System F, using a dictionary passing style a la Haskell.
- Thus, $\mathrm{F}^{\mathcal{G}}$ can be separately compiled.

A Larger Prototype: $\mathcal{G}$

$\mathcal{G}$ adds the following on top of $\mathrm{F}^{\mathcal{G}}$:

- Template argument deduction.
- Function overloading.
- Concept refinement (like **requires**, but flattens out).
- A simple module system with ability to import/export models.
- Parameterized models with constraints.
- A simple form of classes.

We were able to implement most of the STL in $\mathcal{G}$ [Siek 2005].

Ironically, vtables play the role of dictionaries in the implementation of $\mathcal{G}$.

Back to the Dispatching Problem

```
fun copy<Iter1,Iter2> where { InputIterator<Iter1>,
    OutputIterator<Iter2, InputIterator<Iter1>.value> }
(Iter1 first, Iter1 last, Iter result) -> Iter2;

fun copy<Iter1,Iter2> where { RandomAccessIterator<Iter1>,
    OutputIterator<Iter2, InputIterator<Iter1>.value> }
(Iter1 first, Iter1 last, Iter2 result) -> Iter2;

fun merge<Iter1,Iter2,Iter3>
where { InputIterator<Iter1>, InputIterator<Iter2>, ... }
(Iter1 first1, Iter1 last1, Iter2 first2, Iter2 last2, Iter3 result) -> Iter3 {
  ...
  return copy(first2, last2, copy(first1, last1, result));
}
```

The calls to copy always resolve to the version for InputIterator.

The Workaround, Propagate Dispatch to the Top

```
concept CopyRange<I1,I2> {
  fun copy_range(I1,I1,I2) -> I2;
};
model <Iter1,Iter2> where { InputIterator<Iter1>, ... }
CopyRange<Iter1,Iter2> {
  fun copy_range(Iter1 first, Iter1 last, Iter2 result) -> Iter2
    { return copy(first, last, result); }
};
model <Iter1,Iter2> where { RandomAccessIterator<Iter1>, ... }
CopyRange<Iter1,Iter2> {
  fun copy_range(Iter1 first, Iter1 last, Iter2 result) -> Iter2
    { return copy(first, last, result); }
};
fun merge<Iter1,Iter2,Iter3>
where { ..., CopyRange<Iter2,Iter3>, CopyRange<Iter1,Iter3> }
(Iter1 first1, Iter1 last1, Iter2 first2, Iter2 last2, Iter3 result) -> Iter3 {
  ... return copy_range(first2, last2, copy_range(first1, last1, result));
}
```

The Indiana Design: Concepts for C++0x

- By the end of 2004, Stroustrup had not updated his proposal.
- So the Indiana group submitted a design for concepts in January 2005, Concepts for C++0X, N1758.
- This design differed from $\mathcal{G}$ is several respects:
  - There was no hope for separate compilation, but we still wanted type checking to be as modular as possible.
  - We decided to go with two-phase concept-based overload resolution.
  - The implementation did not use dictionaries, but instead resolved concept-indirect calls during compilation.
  - Models are not lexically scoped, but reside in the same namespace as their concept. (More about this later.)
- Doug Gregor began implementing our design in the GNU C++ compiler.

Example Concepts

```
template<typeid T>
concept LessThanComparable {
  bool operator<(T a, T b);
  bool operator>(T a, T b) {return b < a;}
  bool operator<=(T a, T b) {return !(b < a);}
  bool operator>=(T a, T b) {return !(a < b);}
};
template<typeid Iter>
concept InputIterator : EqualityComparable<Iter>,
    CopyConstructible<Iter>, Assignable<Iter>
{
  typename value_type = Iter::value_type;
  typename difference_type = Iter::difference_type;
  typename reference = Iter::reference;
  require Convertible<reference, value_type>;
  Iter& operator++(Iter&);
  reference operator*(Iter);
};
```

Example Models

```
model LessThanComparable<int> { };
model LessThanComparable<float> { };

template<typeid T>
model InputIterator<T*> { };

template<typeid Iter>
  where { BidirectionalIterator<Iter> }
model BidirectionalIterator< reverse_iterator<Iter> > { };
```

Example Generic Algorithm

```
template<typeid InIter, typeid OutIter>
where { InputIterator<InIter>,
        OutputIterator<OutIter, InputIterator<InIter>::value_type> }
OutIter copy(InIter a, InIter b, OutIter out) {
  while (a != b)
    *out++ = *a++;
  return out;
}
```

Pseudo-Signatures

```
class A {};
class B : public A {};

template<typeid T>
concept C {
  A f(T, B);
};

B f(A, A) { return B(); }

model C<A> { };
// This model is OK because B is convertible to A.
```

Modular Type Checking

Did we achieve modular type checking? Not completely, thanks to template specialization.

```
template <class T> struct vector {
  vector(int);
};

template <LessThanComparable T>
void f(T x) { vector<T> v(100); ... }

template <> struct vector<int> {
  // missing constructor
};

int main() { f(4); }
```

The Texas Rebuttal

- By April 2005, Stroustrup and Dos Reis had revised their original proposal.
- This revision is closer to the Indiana in some respects:
  1. It includes **where** clauses for expressing multi-type concepts.
  2. It includes associated types.
- But there are significant differences:
  1. usage patterns
  2. general Boolean expressions allowed in **where** clauses
  3. || and ! operators on constraints
  4. Models declarations are optional unless a concept has associated types.
  5. The proposal did not include a detailed semantics and there was no prototype implementation.

The Compromise at Adobe Systems

- Stepanov invited both teams to a meeting at Adobe.
- The resulting compromise was as follows.
- Pseudo-signatures over usage patterns.
- Include || and ! constraints.
- On the topic of implicit vs. explicit models, the compromise was to provide both by having two kinds of concepts:
  - Concepts that require models (the default).
  - "Auto" concepts that do not require models.

  While this complicated the design, I felt it was a good compromise, balancing convenience and safety.
- Use the semantics from the Indiana design for concept/model matching, type checking constrained templates, and for checking constrained template instantiations.

The Compromise Design, 2006

- Over the next few months, the Indiana and Texas teams wrote up the compromise design [OOPSLA 2006].
- Gregor and Stroustrup wrote a revised proposal for the standards committee: N2042, N2081.
- The keyword **model** became **concept_map** because a survey showed that **model** appears too frequently in C++ programs.

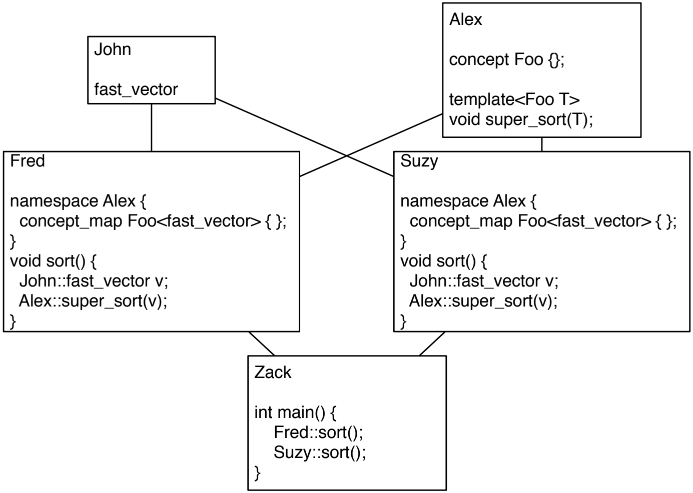
Scoped Concept Maps

As of 2006, the concepts design required concept maps to appear in the same namespace as their concept. E.g.

```
namespace std {
  concept EqualityComparable<typename T> { ... };
}

namespace UserLib {
  class A {};
}
namespace std {
  concept_map EqualityComparable<UserLib::A> { };
}
```

Scoped Concept Maps

So concept maps are essentially global, and anything global is bad for modularity. In particular, consider the following:



```
fast_vector
```

```
concept Foo {};

template<Foo T>
void super_sort(T);
```

```
namespace Alex {
  concept_map Foo<fast_vector> { };
}
void sort() {
  John::fast_vector v;
  Alex::super_sort(v);
}
```

```
namespace Alex {
  concept_map Foo<fast_vector> { };
}
void sort() {
  John::fast_vector v;
  Alex::super_sort(v);
}
```

```
int main() {
    Fred::sort();
    Suzy::sort();
}
```

Scoped Concept Maps, [Siek 2006]

- Transferred the notion of scoped concepts from $\mathcal{G}$.
- Concept maps can be placed in any namespace and their scope is the usual lexical one.

```
namespace AdditiveMonoid {
  concept_map Monoid<int> {
    int binary_op(int x, int y) { return x + y; }
    int identity_elt() { return 0; }
  };
}
namespace MultiplicativeMonoid {
  concept_map Monoid<int> {
    int binary_op(int x, int y) { return x * y; }
    int identity_elt() { return 1; }
  };
}

int main() {
  vector<int> v;
  { using namespace AdditiveMonoid; cout << foldl(v.begin(), v.end()); }
  { using namespace MultiplicativeMonoid; cout << foldl(v.begin(), v.end()); }
}
```

Proposed Wording for Concepts, 2007 and 2008

- Gregor and Stroustrup flesh out the "standardese" for concepts in the revisions: N2193, N2307, N2398.
- Siek and Widman help with the standardese: N2421, N2501, N2520.
- The **where** keyword is changed to **requires** (less frequently used).
- The || constraints are removed for lack of implementation experience.
- The worries concerning the cost of "forwarding functions" is laid to rest. Gregor shows you can implement concepts without them (but lost some modularity in the process).

The Yo-Yo, 2008-2009

- At the San Francisco meeting, September 2008, concepts are voted into C++0x. Hurrah!
- In the following months, there are heated discussions on the C++ standards committee mailing list.
- Concern over the limited implementation experience:
  - The gcc prototype is slow (do difficult to implement congruence closure in gcc)
  - It's missing several features: scoped concept maps, associated templates.
- There is even more concern over "explicit" concepts. Would the average user have to write too many concept maps?

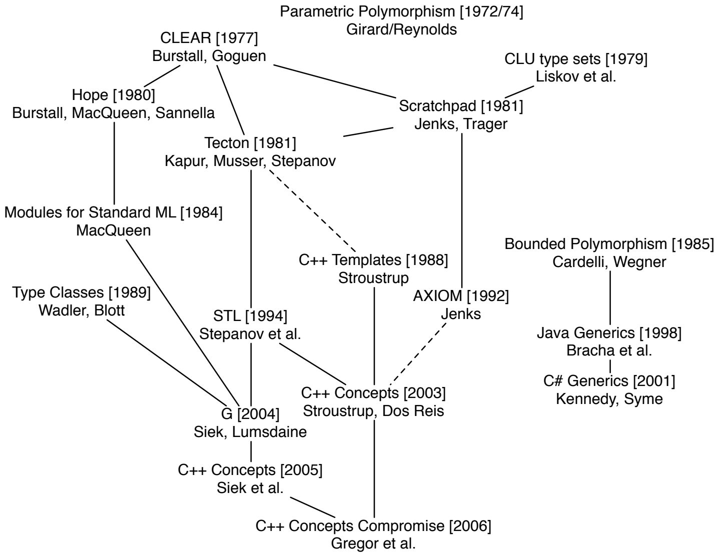
Frankfurt, July 2009

- Stroustrup proposes an alternative design: one kind of concept (implicit), but two kinds of refinement.
- A vote is taken between the following options:
  1. Continue with the current specification of concepts in C++0x.
  2. Make the changes suggested by Stroustrup and retain concepts in C++0x.
  3. Remove concepts from C++0x.
- It was too late for major changes, and without Bjarne supporting the status quo, an overwhelming majority voted to remove concepts (even the Indiana folks).

What's Next?

- Next opportunity for the standardization of concepts is probably 2015.
- One big questions are whether there will be the resources to implement a production quality version of concepts.
- There's some hope that Clang (the front-end for LLVM) will become a good platform. Doug Gregor is currently working on getting Clang up to standards conformance (circa 2003).
- However, it's difficult to convince companies to invest significant resources to these kind of volunteer efforts.

Influences on the “concepts” effort


Parametric Polymorphism [1972/74]
Girard/Reynolds
CLEAR [1977]
Burstall, Goguen
CLU type sets [1979]
Liskov et al.
Hope [1980]
Burstall, MacQueen, Sannella
Scratchpad [1981]
Jenks, Trager
Tecton [1981]
Kapur, Musser, Stepanov
Modules for Standard ML [1984]
MacQueen
Bounded Polymorphism [1985]
Cardelli, Wegner
C++ Templates [1988]
Stroustrup
Type Classes [1989]
Wadler, Blott
AXIOM [1992]
Jenks
STL [1994]
Stepanov et al.
Java Generics [1998]
Bracha et al.
C# Generics [2001]
Kennedy, Syme
C++ Concepts [2003]
Stroustrup, Dos Reis
G [2004]
Siek, Lumsdaine
C++ Concepts [2005]
Siek et al.
C++ Concepts Compromise [2006]
Gregor et al.