A Comparative Study of Language Support for Generic Programming

Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock

Open Systems Laboratory, Indiana University
What is this talk about?

- A Comparative Study of Language Support for Generic Programming
What is this talk about?

• A Comparative Study of Language Support for Generic Programming
• Generic programming is more than List<T>
What is this talk about?

• A Comparative Study of Language Support for Generic Programming
• Generic programming is more than `List<T>`
• Why do language designers make generic programmers suffer?
What is this talk about?

• A Comparative Study of Language Support for Generic Programming

• Generic programming is more than List<T>

• Why do language designers make generic programmers suffer?

• Generic programming is the best thing since sliced bread — a big toaster survey
What this talk is really about

- Evaluate support for generic programming in 6 “mainstream” programming languages
  - C++, SML, Haskell, Eiffel, Java Generics, Generic C#
- Identify language features critical to successful generic programming
- Non-goal: to compare generic programming to other paradigms/programming styles
- Non-goal: to compare the languages with regard to anything other than generic programming
- Results:
  - Object-oriented subtyping is not a good match for constraining type parameters
  - Implicit instantiation is necessary
  - Subtype relationships should not be fixed when a type is defined
  - Type aliases are important
  - Type classes are a close approximation to concepts
Motivation

• Recent interest in generic programming
• A lot of work in C++. Why?
  – Unconstrained genericity?
• How much do safer generic mechanisms of other languages get in the way?
The experiment

- Implementation of a library of generic graph algorithms
  - Modeled after the Boost Graph Library [Siek, Lee, Lumsdaine, A-W, 2001]
- Non-trivial example of generic programming
  - Algorithms that require some associated types of their arguments to be the same (sharing constraints)
  - Algorithms with a lot of parameters
    - Graph type is just one type among many
  - Generic algorithms use other generic algorithms

⇒ Exploits generic programming paradigm even more than STL
Graph algorithms implemented

- Graph search
- Breadth-first search
- Dijkstra’s single source shortest paths
- Prim’s minimum spanning tree
- Johnson’s all-pairs shortest paths
- Bellman-Ford single source shortest paths
Outline

• Introduction
• Experiment
• Generic programming
• Evaluation of language features
• Conclusions
Generic programming is a sub-discipline of computer science that deals with finding abstract representations of efficient algorithms, data structures, and other software concepts, and with their systematic organization. The goal of generic programming is to express algorithms and data structures in a broadly adaptable, interoperable form that allows their direct use in software construction.

[M. Jazayeri, R. Loos, D. Musser, and A. Stepanov, 1998]

- Algorithms separated from specific data structures
  - An algorithm is not tied to a specific graph type

- Minimal assumptions about input types
  - What is the most general category of data structures that an algorithm can use?

- Automatic dispatching to most efficient implementation
  - More than minimal capabilities can allow a faster/better implementation

- No abstraction penalty
  - Algorithms should be as efficient as those written for specific data types
Generic programming components

- Generic algorithms
  - Constrain arguments using concepts

- Concepts
  - Requirements on data types used with a particular algorithm or set of algorithms
  - Give information on required operations that the algorithm(s) need
  - Can refine other concepts (requires conformance to the other concepts)

- Major components:
  - Valid expressions/function requirements:
    - Functions that must exist and their signatures
  - Associated types
    - E.g., value type of an iterator, vertex type of a graph
  - Semantic constraints:
    - Preconditions, postconditions, invariants, complexity guarantees

- Data structures
  - Model concepts (satisfy all of the constraints in a concept)
... generic programming components

<table>
<thead>
<tr>
<th>Role</th>
<th>C++</th>
<th>ML</th>
<th>Haskell</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Generic algorithm</strong></td>
<td>function template</td>
<td>functor</td>
<td>polymorphic func.</td>
</tr>
<tr>
<td><strong>Concept</strong></td>
<td>⟨documentation⟩</td>
<td>signature</td>
<td>inheritance (⇒)</td>
</tr>
<tr>
<td><strong>Refinement</strong></td>
<td>⟨documentation⟩</td>
<td>include</td>
<td>instance</td>
</tr>
<tr>
<td><strong>Modeling</strong></td>
<td>⟨documentation⟩</td>
<td>⟨implicit⟩</td>
<td>context (⇒)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Role</th>
<th>Eiffel</th>
<th>Java generics</th>
<th>Generic C#</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Generic algorithm</strong></td>
<td>generic class</td>
<td>generic method</td>
<td>generic method</td>
</tr>
<tr>
<td><strong>Concept</strong></td>
<td>inherit</td>
<td>interface</td>
<td>inherit (:)</td>
</tr>
<tr>
<td><strong>Refinement</strong></td>
<td>inherit</td>
<td>extends</td>
<td>inherit (:)</td>
</tr>
<tr>
<td><strong>Modeling</strong></td>
<td>conformance (→)</td>
<td>extends</td>
<td>where</td>
</tr>
<tr>
<td><strong>Constraint</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Representing models relationships

- Different languages express modeling in different ways
  - Structural conformance (C++, SML): a type which satisfies the constraints of a concept is automatically a model of the concept
  - Named conformance (Haskell [type classes]; Eiffel, Java, C# [interfaces]): a type must be explicitly declared to model a concept
    * Avoids problems with *accidental conformance*
    * Can either be declared inside the type itself or in a separate declaration
Generic programming example

- Dot product of two sequences
- Sequences represented using ranges (enumerators)
- Elements of sequences must be multipliable and the result of this multiplication must be addable with `sum`.

Illustrative example:
- Typical, and easily written, generic algorithm in the spirit of the STL
- Challenging for many other programming paradigms
  - Use of associated types
  - Constraints on associated types
  - Associated types of different parameters must be related
  - Multi-parameter concepts

- Note that the example is written in pseudocode, using an imaginary language providing direct support for concepts
Dot product algorithm

forall (Seq1, Seq2, SumType)
SumType dot_product(Seq1 s1, Seq2 s2, SumType sum)
where Seq1 models Range as S1, Seq2 models Range as S2,
(S1.value_type, S2.value_type) models Multipliable as Mult,
(SumType, Mult.result_type) models Addable as Add,
Add.result_type = SumType,
SumType models Assignable

while (true)
  case (S1.is_empty(s1), S2.is_empty(s2)) of
    (true, true) → return sum;
    (true, false), (false, true) → raise "Sequences not of equal length";
    (false, false) →
      sum = Add.add(sum, Mult.mult(S1.first(s1), S2.first(s2)));
      S1.next(s1); S2.next(s2);
Concept, models declaration, and call

```cpp
concept Range[R] {
    type value_type;
    value_type first(R);
    bool is_empty(R);
    void next(R&);
}

model Range[my_range] {
    type value_type = ...;
    value_type first(R ran) { ... }
    bool is_empty(R ran) { ... }
    void next(R& ran) { ... }
}

dot_product(range1, range2, 0);
```
## Language comparison

<table>
<thead>
<tr>
<th>Feature</th>
<th>C++</th>
<th>ML</th>
<th>Haskell</th>
<th>Eiffel</th>
<th>Java</th>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-type concepts</td>
<td>-</td>
<td>🟢</td>
<td>🟢*</td>
<td>🟡</td>
<td>🟢</td>
<td>🟡</td>
</tr>
<tr>
<td>Multiple constraints</td>
<td>-</td>
<td>🟢</td>
<td>🟢</td>
<td>🟡</td>
<td>🟢</td>
<td>🟡</td>
</tr>
<tr>
<td>Associated type access</td>
<td>🟢</td>
<td>🟢</td>
<td>🟢</td>
<td>🟡</td>
<td>🟢</td>
<td>🟡</td>
</tr>
<tr>
<td>Retroactive modeling</td>
<td>🟢</td>
<td>🟢</td>
<td>🟢</td>
<td>🟡</td>
<td>🟢</td>
<td>🟡</td>
</tr>
<tr>
<td>Type aliases</td>
<td>🟢</td>
<td>🟢</td>
<td>🟢</td>
<td>🟡</td>
<td>🟢</td>
<td>🟡</td>
</tr>
<tr>
<td>Separate compilation</td>
<td>🟡</td>
<td>🟢</td>
<td>🟢</td>
<td>🟡</td>
<td>🟢</td>
<td>🟡</td>
</tr>
<tr>
<td>Implicit instantiation</td>
<td>🟢</td>
<td>🟢</td>
<td>🟢</td>
<td>🟡</td>
<td>🟢</td>
<td>🟡</td>
</tr>
<tr>
<td>Concise syntax</td>
<td>🟢</td>
<td>🟢</td>
<td>🟢</td>
<td>🟡</td>
<td>🟢</td>
<td>🟡</td>
</tr>
</tbody>
</table>

*Using the multi-parameter type class extension to Haskell 98.
†Planned language additions.
‡Planned for inclusion in Whidbey release of C#.
Multi-type concepts

<table>
<thead>
<tr>
<th></th>
<th>C++</th>
<th>ML</th>
<th>Haskell</th>
<th>Eiffel</th>
<th>Java</th>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>-</td>
<td>●</td>
<td>●*</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
</tbody>
</table>

- VectorSpace[VectorType, ScalarType]
- Assignable[T, U]
- Why? It’s just not general enough otherwise
- Haskell: multi-parameter type classes
- Eiffel, Java, C#: interfaces are constraints on only one type
  - Can use type parameters on constraints $\implies$ constraint explosion
- Not a major problem in implementing the graph library
## Multiple constraints

<table>
<thead>
<tr>
<th>C++</th>
<th>ML</th>
<th>Haskell</th>
<th>Eiffel</th>
<th>Java</th>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>$\circ$</td>
<td>![Filled Circle]</td>
<td>![Empty Circle] $\dagger$</td>
<td>![Filled Circle]</td>
<td>![Filled Circle] $\dagger$</td>
</tr>
</tbody>
</table>

```plaintext
forall (G, ...) 
void bfs(G g, ...) 
where G models VertexListGraph, 
      G models IncidenceGraph ...
```

- Frequently used, no good way to emulate
  - `VertexListAndIncidenceGraph` or `IncidenceAndVertexListGraph`?
- ML: in order to constrain parameters multiple times, they must be repeated in the structure
- C++: no concepts, no problems
- Other languages: either have it, or are planning to add it
Associated type access

<table>
<thead>
<tr>
<th>C++</th>
<th>ML</th>
<th>Haskell</th>
<th>Eiffel</th>
<th>Java</th>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

- Types related to the main type(s) of a concept
  - Functionally dependent, can thus be looked up based on the main types
- A way to express relations between type and encapsulate these relations
- C++: member typedefs or traits classes, ML: types in signatures/structures
- Haskell, Eiffel, C#, Java: extra type parameters
public class breadth_first_search {
    public static void go<G, Vertex, Edge, VertexIterator, OutEdgeIterator, ColorMap, Visitor>(G g, Vertex s, ColorMap c, Visitor vis)
        where G: VertexListAndIncidenceGraph<
            Vertex, Edge, VertexIterator,
            OutEdgeIterator>,
        Edge: GraphEdge<Vertex>,
        VertexIterator: IEnumerable<Vertex>,
        OutEdgeIterator: IEnumerable<Edge>,
        ColorMap: ReadWriteMap<Vertex, ColorValue>,
        Visitor: BFSVisitor<G, Vertex, Edge>;
}
... associated type access

```java
public class breadth_first_search {
    public static void go<G, Vertex, Edge, VertexIterator, OutEdgeIterator, ColorMap, Visitor>(G g, G.Vertex s, ColorMap c, Visitor vis)
        where G: VertexListAndIncidenceGraph,<
    Vertex, Edge, VertexIterator,
    OutEdgeIterator>,
    Edge: GraphEdge<Vertex>,
    VertexIterator: IEnumerable<Vertex>,
    OutEdgeIterator: IEnumerable<Edge>,
    ColorMap: ReadWriteMap<G.Vertex, ColorValue>,
    Visitor: BFSVisitor<G, Vertex, Edge>;
}
```

- Code in red can be removed, code in blue must be added
Retroactive modeling

**Library A:**

```java
public interface VertexListGraph<...> where ...
    { 
    VertexIterator vertices(); 
    int num_vertices(); 
    }
public interface EdgeListGraph<...> where ...
    { 
    EdgelIterator edges(); 
    }
public class adjacency_list 
    : VertexListGraph<...>, 
    EdgeListGraph<...> { ... }
```

**Library B:**

```java
public interface VertexNumberGraph<...> where ...
    { 
    int num_vertices(); 
    }
class bellman_ford 
    { 
    public static void go<G, ...>(g, ...)
        where G: VertexNumberGraph<...>,
        A.EdgeListGraph<...>
        { ... }
    }
```

**Library C:**

```java
A.adjacency_list g;
// Problem: A.adjacency_list does not inherit from VertexNumberGraph
B.bellman_ford.go<A.adjacency_list, ...>(g, ...);
```
... retroactive modeling

<table>
<thead>
<tr>
<th>C++</th>
<th>ML</th>
<th>Haskell</th>
<th>Eiffel</th>
<th>Java</th>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
</tbody>
</table>

- Allows new modeling or refinement relationships to be added after a library is defined
- Important for extending libraries without breaking compatibility with old code
- C++: no direct support for concepts
- ML: structural conformance, so no problem
- Haskell: allows instance declarations to be given outside of a type definition
- Object-oriented languages: inheritance fixed when a type is defined
  - Known problem in object-oriented programming, even without generics
  - Solutions: aspect-oriented programming, Half & Half, etc.
Type aliases

<table>
<thead>
<tr>
<th></th>
<th>C++</th>
<th>ML</th>
<th>Haskell</th>
<th>Eiffel</th>
<th>Java</th>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
</tbody>
</table>

- A shorthand name for a particular type
- Useful in its own right as an abstraction mechanism.
- Crucial in GP to prevent repetition of complicated type names
- No type aliases + no implicit instantiation → unnecessarily verbose code
• From Dijkstra’s shortest paths algorithm (C#):

```csharp
dijkstra_visitor<G, 
mutable_queue<Vertex, indirect_cmp<Vertex, Distance, DistanceMap, DistanceCompare>>, 
WeightMap, PredecessorMap, DistanceMap, DistanceCombine, DistanceCompare, 
Vertex, Edge, Distance>
bfs_vis = 
    new dijkstra_visitor<G, 
mutable_queue<Vertex, indirect_cmp<Vertex, Distance, DistanceMap, DistanceCompare>>, 
WeightMap, PredecessorMap, DistanceMap, DistanceCombine, DistanceCompare, Vertex, Edge, Distance>();

graph_search.go<G, Vertex, Edge, VertexIterator, OutEdgeIterator, hash_map<Vertex, ColorValue>>, 
mutable_queue<Vertex, indirect_cmp<Vertex, Distance, DistanceMap, DistanceCompare>>, 
dijkstra_visitor<G, 
mutable_queue<Vertex, indirect_cmp<Vertex, Distance, DistanceMap, DistanceCompare>>, 
WeightMap, PredecessorMap, DistanceMap, DistanceCombine, DistanceCompare, Vertex, Edge, Distance> > (g, s, color, Q, bfs_vis);
```
... type aliases

- From Dijkstra’s shortest paths algorithm (C#):

```csharp
    dijkstra_visitor<G,
    mutable_queue<Vertex, indirect_cmp<Vertex, Distance, DistanceMap, DistanceCompare> >,
    WeightMap, PredecessorMap, DistanceMap, DistanceCombine, DistanceCompare,
    Vertex, Edge, Distance>
    bfs_vis =
    new dijkstra_visitor<
    G, mutable_queue<Vertex, indirect_cmp<Vertex, Distance, DistanceMap, DistanceCompare> >,
    WeightMap, PredecessorMap, DistanceMap, DistanceCombine, DistanceCompare, Vertex, Edge,
    Distance>();

    graph_search.go<
    G, Vertex, Edge, VertexIterator, OutEdgeIterator, hash_map<Vertex, ColorValue>,
    mutable_queue<Vertex, indirect_cmp<Vertex, Distance, DistanceMap, DistanceCompare> >,
    dijkstra_visitor<G,
    mutable_queue<Vertex, indirect_cmp<Vertex, Distance, DistanceMap, DistanceCompare> >,
    WeightMap, PredecessorMap, DistanceMap, DistanceCombine, DistanceCompare, Vertex, Edge,
    Distance> > (g, s, color, Q, bfs_vis);
```

- The type in **blue** is repeated several times (occurrences in **red**).
... type aliases

- From Dijkstra’s shortest paths algorithm (C#):

```csharp
dijkstra_visitor<G,
    mutable_queue<Vertex, indirect_cmp<Vertex, Distance, DistanceMap, DistanceCompare> >,
    WeightMap, PredecessorMap, DistanceMap, DistanceCombine, DistanceCompare,
    Vertex, Edge, Distance>

bfs_vis =
    new dijkstra_visitor<
        G, mutable_queue<Vertex, indirect_cmp<Vertex, Distance, DistanceMap, DistanceCompare> >,
        WeightMap, PredecessorMap, DistanceMap, DistanceCombine, DistanceCompare, Vertex, Edge,
        Distance>();

dijkstra_visitor<G,
    mutable_queue<Vertex, indirect_cmp<Vertex, Distance, DistanceMap, DistanceCompare> >,
    WeightMap, PredecessorMap, DistanceMap, DistanceCombine, DistanceCompare, Vertex, Edge,
    Distance> > (g, s, color, Q, bfs_vis);
```

- The type in blue is repeated several times (occurrences in red)
From Dijkstra’s shortest paths algorithm (C#):

```csharp
typedef mutable_queue<Vertex, indirect_cmp<Vertex, Distance, DistanceMap, DistanceCompare>> as queue_type;

typedef dijkstra_visitor<G, queue_type, WeightMap, PredecessorMap, DistanceMap, DistanceCombine, DistanceCompare, Vertex, Edge, Distance> as visitor_type;

visitor_type bfs_vis = new visitor_type();

graph_search.go<G, Vertex, Edge, VertexIterator, OutEdgeIterator, hash_map<Vertex, ColorValue>, queue_type, visitor_type> (g, s, color, Q, bfs_vis);
```

Code after replacing repeated types with aliases
Separate compilation

<table>
<thead>
<tr>
<th>C++</th>
<th>ML</th>
<th>Haskell</th>
<th>Eiffel</th>
<th>Java</th>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

- Lack of separate type checking → errors in generics caught too late
  - May appear only at a particular instantiation
- Separate compilation allows shipping of generic libraries as binaries
- Different levels, with different trade-offs
  - C++: completely non-separate type checking and compilation
  - Eiffel: separate compilation, but full type safety requires whole-program analysis
  - C#: completely separate type checking and byte-code compilation, instantiation at load time
  - Others: fully separate type-checking and compilation
- Separate type checking is important; separate compilation is a business decision
Implicit instantiation

<table>
<thead>
<tr>
<th>C++</th>
<th>ML</th>
<th>Haskell</th>
<th>Eiffel</th>
<th>Java</th>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○†</td>
</tr>
</tbody>
</table>

- Leads to verbosity by itself (ML)
- With no associated type representation (Eiffel, evaluated version of C#):
  - Excessive verbosity (ridiculous if no type aliases)
  - Exposes internal types of data structures to users (code in red or blue)

```csharp
breadth_first_search<
    adjacency_list, int, adj_list_edge<int>, // edge type
    IEnumerable<int>, // vertex iterator type
    IEnumerable<adj_list_edge<int>>, // out edge iterator type
    ColorMap,
    my_bfs_visitor<adjacency_list, int, adj_list_edge<int>> > >
(graph, src_vertex, color_map, visitor);
```
Concise syntax

<table>
<thead>
<tr>
<th>C++</th>
<th>ML</th>
<th>Haskell</th>
<th>Eiffel</th>
<th>Java</th>
<th>C#</th>
</tr>
</thead>
<tbody>
<tr>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>○</td>
<td>○</td>
</tr>
</tbody>
</table>

- Criteria: syntax of generic function definitions and calls
  - Definitions must not contain excessive amounts of redundant information
  - Calling a (constrained) generic function should be the same as calling a normal, non-generic or non-constrained function

- C++, Haskell: generic functions are relatively simple to write, calling a generic function is exactly like calling a non-generic function

- ML: call syntax different and more verbose for constrained generics (functors)

- OO languages: many redundant constraints and type parameters must be written in generic algorithms
  - Eiffel, C# (evaluated version): explicit type arguments to generics
OO subtyping is not modeling

- Refining concept includes all constraints from the refined concept
- Subinterface includes all methods from superinterface, but not parameter constraints
- Many redundant constraints, and this only gets worse as concept hierarchy gets deeper (scalability issue)

Shown on the next slide: red marks redundant constraints, blue marks necessary constraints
... OO subtyping is not modeling

```java
public interface VertexListGraph<
    Vertex, VertexIter>
    where VertexIter: IEnumerable<
    Vertex>
    {
    VertexIter vertices();
    int num_vertices();
    }

public interface IncidenceGraph<
    Vertex, Edge, OutEdgeIter>
    where Edge: GraphEdge<
    Vertex>, OutEdgeIter: IEnumerable<
    Edge>
    {
    OutEdgeIter out_edges(Vertex v);
    int out_degree(Vertex v);
    }

public interface VertexListAndIncidenceGraph
    <Vertex, Edge, VertexIter, OutEdgeIter>
    : VertexListGraph<
    Vertex, VertexIter>,
    IncidenceGraph<
    Vertex, Edge, OutEdgeIter>
    where Edge: GraphEdge<
    Vertex>,
    VertexIter: IEnumerable<
    Vertex>,
    OutEdgeIter: IEnumerable<
    Edge> {}
... OO subtyping is not modeling

```java
public static void bfs<G, Vertex, Edge, VertexIterator, OutEdgeIterator, ColorMap, Visitor>(G g, Vertex s, ColorMap c, Visitor vis)
   where G: VertexListAndIncidenceGraph<Vertex, Edge, VertexIterator, OutEdgeIterator>,
       Edge: GraphEdge<Vertex>,
       VertexIterator: IEnumerable<Vertex>,
       OutEdgeIterator: IEnumerable<Edge>,
       ColorMap: ReadWriteMap<Vertex, ColorValue>,
       Visitor: BFSVisitor<G, Vertex, Edge>;
```

- Many redundant constraints (in red) are required here, even though `VertexListAndIncidenceGraph` already includes the same constraints

- Lack of associated types exacerbates this problem
  - With proper support for associated types, only type parameters in green need to be present
The next sequence of slides shows how much the implementation of the Dijkstra’s shortest path algorithm could be simplified if C# supported the key features of generic programming we identified. For each feature, there is a slide that identifies the feature, followed by a slide that marks code that can be removed or simplified:

- **Red** is for code that can be removed
- **Blue** is for code that can be changed, usually simplified.

Next follows a slide where the removal and appropriate changes have been made. The first slide shows the effect of differences between the Gyro implementation and its specification.
public class dijkstra_shortest_paths {
    public static void go(<
        GraphT, Vertex, Edge, VertexIterator, OutEdgeIterator,
        DegreeSizeType, WeightMap, DistanceMap, Distance,
        PredecessorMap, DistanceCombine, DistanceCompare>
    (GraphT g, Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
        DistanceCombine combine, Distance int, Distance zero) {
        for (IEnumerator<Vertex> ui = g.vertices().GetEnumerator(); ui.MoveNext();)
            {
            Vertex u = ui.Current;
            ((ReadWritePropertyMap<Vertex,Distance>)distance)
                [u] = inf;
            ((ReadWritePropertyMap<Vertex,Vertex>)predecessor)
                [u] = u;
        }
        ((ReadWritePropertyMap<Vertex,Distance>)distance)
            [s] = zero;
        indirect_cmp<_Vertex, Distance, DistanceMap, DistanceCompare> icmp = new indirect_cmp<_Vertex, Distance, DistanceMap, DistanceCompare> (distance, compare);
        mutable_queue<_Vertex, VerticesSizeType, indirect_cmp<_Vertex, Distance, DistanceMap, DistanceCompare>> Q = new mutable_queue<_Vertex, VerticesSizeType, indirect_cmp<_Vertex, Distance, DistanceMap, DistanceCompare>> (icmp);
        dijkstra_bfs_visitor<_GraphT, mutable_queue<_Vertex, VerticesSizeType, indirect_cmp<_Vertex, Distance, DistanceMap, DistanceCompare>>> bfs_vis = new dijkstra_bfs_visitor<_GraphT, mutable_queue<_Vertex, VerticesSizeType, indirect_cmp<_Vertex, Distance, DistanceMap, DistanceCompare>>> (g, s, weight, predecessor, distance, combine, compare, zero);
        simple_property_map<_Vertex, ColorValue> color = new simple_property_map<_Vertex, ColorValue> ();
        // Initialize color map
        for (IEnumerator<Vertex> ui = g.vertices().GetEnumerator(); ui.MoveNext();)
            {
            Vertex v = ui.Current;
            color[v] = ColorValue.white;
        }
        breadth_first_visit.go(<
            GraphT, Vertex, Edge, VertexIterator, OutEdgeIterator,
            DegreeSizeType, WeightMap, DistanceMap, Distance,
            PredecessorMap, DistanceCombine, DistanceCompare>
        (g, s, bfs_vis, color, Q);
    }
}
```java
public class dijkstra_shortest_paths {
    public static void go<
        GraphT, Vertex, Edge, VertexIterator, OutEdgeIterator,
        DegreeSizeType, WeightMap, DistanceMap, Distance,
        PredecessorMap, DistanceCombine, DistanceCompare>
    (GraphT g, Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
     DistanceCombine combine, Distance inf, Distance zero)
    {
        where GraphT: VertexListAndIncidenceGraph<
            Vertex, Edge, VertexIterator, VerticesSizeType, OutEdgeIterator,
            DegreeSizeType, WeightMap, DistanceMap, Distance, PredecessorMap, DistanceCombine, DistanceCompare>

        (for (IEnumerator<
                    Vertex> ui = g.vertices().GetEnumerator(); ui.MoveNext();)
        {
            Vertex u = ui.Current;
            ((ReadWritePropertyMap<
                        Vertex, Distance>) distance)[u] = inf;
            ((ReadWritePropertyMap<
                        Vertex, Vertex>) predecessor)[u] = u;
        })

        ((ReadWritePropertyMap<
                        Vertex, Distance>) distance)[s] = zero;

        indirect_cmp<
            Vertex, Distance, DistanceMap, DistanceCompare>
        icmp =
            new indirect_cmp<
                Vertex, Distance, DistanceMap, DistanceCompare>(distance, compare);

        mutable_queue<
            Vertex, VerticesSizeType, indirect_cmp<
                Vertex, Distance, DistanceMap, DistanceCompare>> Q =
            new mutable_queue<
                Vertex, VerticesSizeType, indirect_cmp<
                    Vertex, Distance, DistanceMap, DistanceCompare>>(icmp);

        dijkstra_bfs_visitor<
            GraphT, mutable_queue<
                Vertex, VerticesSizeType, indirect_cmp<
                    Vertex, Distance, DistanceMap, DistanceCompare>>,>
        bfs_vis =
            new dijkstra_bfs_visitor<
                GraphT, mutable_queue<
                    Vertex, VerticesSizeType, indirect_cmp<
                        Vertex, Distance, DistanceMap, DistanceCompare>>,>
            (Q, weight, predecessor, distance, combine, compare, zero);

        simple_property_map<
            Vertex, ColorValue>
        color =
            new simple_property_map<
                Vertex, ColorValue>(v, ColorValue.white);
    }
}

// Initialize color map
for (IEnumerator<
            Vertex> ui = g.vertices().GetEnumerator(); ui.MoveNext();)
{
    Vertex v = ui.Current;
    color[v] = ColorValue.white;
}

breadth_first_visit.go<
    GraphT, Vertex, Edge, VertexIterator, VerticesSizeType, OutEdgeIterator,
    DegreeSizeType, WeightMap, DistanceMap, Distance,
    PredecessorMap, DistanceCombine, DistanceCompare>
(g, s, bfs_vis, color, Q);
```
public class dijkstra_shortest_paths {
    public static void go<
        GraphT, Vertex, Edge, VertexIterator, VerticesSizeType, OutEdgeIterator,
        DegreeSizeType, WeightMap, DistanceMap, Distance,
        PredecessorMap, DistanceCombine, DistanceCompare>
    (GraphT g, Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
    DistanceCombine combine, Distance int, Distance zero) {
        foreach (Vertex u in g.vertices()) {
            distance [u] = int;
            predecessor [u] = u;
        }
        distance [s] = zero;
    indirect cmp <Vertex, Distance, DistanceMap, DistanceCompare> icmp = new indirect cmp <Vertex, Distance, DistanceMap, DistanceCompare> (distance, compare);
    mutable queue <Vertex, VerticesSizeType, indirect cmp <Vertex, Distance, DistanceMap, DistanceCompare>> Q = new mutable queue <Vertex, VerticesSizeType, indirect cmp <Vertex, Distance, DistanceMap, DistanceCompare>> (icmp);
    dijkstra bfs visitor <GraphT, mutable queue <Vertex, VerticesSizeType, indirect cmp <Vertex, Distance, DistanceMap, DistanceCompare>>>
    go<
        GraphT, Vertex, Edge, VertexIterator, VerticesSizeType, OutEdgeIterator,
        DegreeSizeType, WeightMap, DistanceMap, Distance,
        PredecessorMap, DistanceCombine, DistanceCompare>
    (g, s, bfs visitor, weight, predecessor, distance, combine, compare, zero);
    simple property map <Vertex, ColorValue> color = new simple property map <Vertex, ColorValue> ();
    // Initialize color map
    foreach (Vertex v in g.vertices()) { color[v] = ColorValue.white; }
    breadth first visit go<
        GraphT, Vertex, Edge, VertexIterator, VerticesSizeType, OutEdgeIterator, DegreeSizeType,
        dijkstra bfs visitor>
    (g, s, bfs visitor, weight, predecessor, distance, combine, compare, zero);
}
Dijkstra shortest paths

• Next: Lack of implicit instantiation

• Implicit instantiation is a proposed addition to C# generics
public class dijkstra_shortest_paths {
    public static void go<
        GraphT, Vertex, Edge, VertexIterator, VerticesSizeType, OutEdgeIterator,
        DegreeSizeType, WeightMap, DistanceMap, Distance,
        PredecessorMap, DistanceCombine, DistanceCompare>
            (GraphT g, Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
                DistanceCombine combine, Distance int, Distance zero)
    {
        foreach (Vertex u in g.vertices()) {
            distance [u] = int;
            predecessor [u] = u;
        }
        distance [s] = zero;
        indirect_cmp <Vertex, Distance, DistanceMap, DistanceCompare> icmp = new indirect_cmp <Vertex, Distance, DistanceMap, DistanceCompare> (distance, compare);
        mutable_queue <Vertex, VerticesSizeType, indirect_cmp <Vertex, Distance, DistanceMap, DistanceCompare> > Q = new mutable_queue <Vertex, VerticesSizeType, indirect_cmp <Vertex, Distance, DistanceMap, DistanceCompare> > (icmp);
        dijkstra_bfs_visitor <GraphT, mutable_queue <Vertex, VerticesSizeType, indirect_cmp <Vertex, Distance, DistanceMap, DistanceCompare> >, WeightMap, PredecessorMap, DistanceMap, DistanceCombine, DistanceCompare, Vertex, Edge, Distance> bfs_vis = new dijkstra_bfs_visitor <GraphT, mutable_queue <Vertex, VerticesSizeType, indirect_cmp <Vertex, Distance, DistanceMap, DistanceCompare> >, WeightMap, PredecessorMap, DistanceMap, DistanceCombine, DistanceCompare, Vertex, Edge, Distance> (Q, weight, predecessor, distance, combine, compare, zero);
        simple_property_map <Vertex, ColorValue> color = new simple_property_map <Vertex, ColorValue> () {
            color(v) = ColorValue.white;
        }
    breadth_first_visit.go<
        GraphT, Vertex, Edge, VertexIterator, VerticesSizeType, OutEdgeIterator, DegreeSizeType,
        dijkstra_bfs_visitor<
            GraphT,
            mutable_queue <Vertex, VerticesSizeType, indirect_cmp <Vertex, Distance, DistanceMap, DistanceCompare> >, WeightMap, PredecessorMap, DistanceMap, DistanceCombine, DistanceCompare, Vertex, Edge, Distance>,
            simple_property_map <Vertex,ColorValue>,
            mutable_queue <Vertex,VerticesSizeType,indirect_cmp <Vertex,Distance,DistanceMap,DistanceCompare> >,
            (g, s, bfs_vis, color, Q)}
public class dijkstra_shortest_paths {
    public static void go<
        GraphT, Vertex, Edge, VertexIterator, VerticesSizeType, OutEdgeIterator,
        DegreeSizeType, WeightMap, DistanceMap, Distance,
        PredecessorMap, DistanceCombine, DistanceCompare>
        (GraphT g, Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
            DistanceCombine combine, Distance inf, Distance zero)
            where GraphT: VertexListAndIncidenceGraph<
                Vertex, Edge, VertexIterator, VerticesSizeType, OutEdgeIterator, DegreeSizeType>,
                Edge: GraphEdge<
                    Vertex>,
                VertexIterator: IEnumerable<
                    Vertex>,
                OutEdgeIterator: IEnumerable<
                    Edge>,
                WeightMap: ReadablePropertyMap<
                    Edge, Distance>,
                DistanceMap: ReadWritePropertyMap<
                    Vertex, Distance>,
                PredecessorMap: ReadWritePropertyMap<
                    Vertex, Vertex>,
                DistanceCombine: BinaryFunction<
                    Distance>,
                DistanceCompare: StrictWeakOrdering<
                    Distance>
            )
            {
                foreach (Vertex u in g.vertices()) {
                    distance[u] = inf;
                    predecessor[u] = u;
                }
                distance[s] = zero;

                indirect_cmp<
                    Vertex, Distance, DistanceMap, DistanceCompare>
                    icmp = new indirect_cmp<
                        Vertex, Distance, DistanceMap, DistanceCompare>
                        (distance, compare);

                mutable_queue<
                    Vertex, VerticesSizeType, indirect_cmp<
                        Vertex, Distance, DistanceMap, DistanceCompare>> Q
                    = new mutable_queue<
                        Vertex, VerticesSizeType, indirect_cmp<
                            Vertex, Distance, DistanceMap, DistanceCompare>>
                        (icmp);

                dijkstra_bfs_visitor<
                    GraphT, mutable_queue<
                        Vertex, VerticesSizeType, indirect_cmp<
                            Vertex, Distance, DistanceMap, DistanceCompare>>, WeightMap, PredecessorMap, DistanceMap, DistanceCombine, DistanceCompare, Vertex, Edge, Distance>
                    bfs_vis
                    = new dijkstra_bfs_visitor<
                        GraphT, mutable_queue<
                            Vertex, VerticesSizeType, indirect_cmp<
                                Vertex, Distance, DistanceMap, DistanceCompare>>, WeightMap, PredecessorMap, DistanceMap, DistanceCombine, DistanceCompare, Vertex, Edge, Distance>
                        (Q, weight, predecessor, distance, combine, compare, zero);

                simple_property_map<
                    Vertex, ColorValue>
                    color = new simple_property_map<
                        Vertex, ColorValue>
                        ();

                // Initialize color map
                foreach (Vertex v in g.vertices()) {
                    color[v] = ColorValue.white;
                }

                breadth_first_visit<
                    GraphT, Vertex, Edge, VertexIterator, VerticesSizeType, OutEdgeIterator,
                    DegreeSizeType, WeightMap, DistanceMap, Distance,
                    PredecessorMap, DistanceCombine, DistanceCompare>
                    (g, s, bfs_vis, color, Q);
            }
    }
Dijkstra shortest paths

- Next: Lack of a type aliasing mechanism
public class dijkstra_shortest_paths {
    public static void go<
        GraphT, Vertex, Edge, VertexIterator, OutEdgeIterator,
        DegreeSizeType, WeightMap, DistanceMap, Distance,
        PredecessorMap, DistanceCombine, DistanceCompare >
    (GraphT g, Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
    DistanceCombine combine, Distance inf, Distance zero)
    where GraphT: VertexListAndIncidenceGraph<
        Vertex,Edge,VertexIterator,VerticesSizeType,OutEdgeIterator,DegreeSizeType >,
        Edge: GraphEdge<
            Vertex",
        VertexIterator: IEnumerable<
            Vertex >,
        OutEdgeIterator: IEnumerable<
            Edge >,
        WeightMap: ReadablePropertyMap<
            Edge,Distance >,
        DistanceMap: ReadWritePropertyMap<
            Vertex,Distance >,
        PredecessorMap: ReadWritePropertyMap<
            Vertex,Vertex >,
        DistanceCombine: BinaryFunction<
            Distance >,
        DistanceCompare: StrictWeakOrdering<
            Distance >
    )
    {
        foreach (Vertex u in g.vertices()) {
            distance [u] = inf;
            predecessor [u] = u;
        }
        distance [s] = zero;
        indirect cmp<
            Vertex, Distance, DistanceMap, DistanceCompare > icmp = new indirect cmp<
            Vertex, Distance, DistanceMap, DistanceCompare > (distance, compare);
        mutable_queue<
            Vertex, VerticesSizeType, indirect cmp<
            Vertex, Distance, DistanceMap, DistanceCompare > > Q
        = new mutable_queue<
            Vertex, VerticesSizeType, indirect cmp<
            Vertex, Distance, DistanceMap, DistanceCompare > > (icmp);
        dijkstra_bfs_visitor<
            GraphT, mutable_queue<
            Vertex, VerticesSizeType, indirect cmp<
            Vertex, Distance, DistanceMap, DistanceCompare > >,
            WeightMap, PredecessorMap, DistanceMap, DistanceCombine, DistanceCompare, Vertex, Edge, Distance
        ) bfs_vis
        = new dijkstra_bfs_visitor<
            GraphT, mutable_queue<
            Vertex, VerticesSizeType, indirect cmp<
            Vertex, Distance, DistanceMap, DistanceCompare > >,
            WeightMap, PredecessorMap, DistanceMap, DistanceCombine, DistanceCompare, Vertex, Edge, Distance
        > (Q, weight, predecessor, distance, combine, compare, zero);
        simple_property_map<
            Vertex, ColorValue > color = new simple_property_map<
            Vertex, ColorValue > () ;
        // Initialize color map
        foreach (Vertex v in g.vertices()) {
            color [v] = ColorValue.white;
        }
        breadth_first_visit <
        GraphT, Vertex, Edge, VertexIterator, VerticesSizeType, OutEdgeIterator, DegreeSizeType, WeightMap, DistanceMap, Distance,
        PredecessorMap, DistanceCombine, DistanceCompare >
    (g, s, bfs_vis, color, Q);
    }
}
public class dijkstra_shortest_paths {
    public static void go<
        GraphT, Vertex, Edge, VertexIterator, VerticesSizeType, OutEdgeIterator,
        DegreeSizeType, WeightMap, DistanceMap, Distance,
        PredecessorMap, DistanceCombine, DistanceCompare>
        (GraphT g, Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare, 
        DistanceCombine combine, Distance inf, Distance zero) {
        foreach (Vertex u in g.vertices()) {
            distance [u] = inf;
            predecessor [u] = u;
        }
        distance [s] = zero;
        typedef indirect<cmp <Vertex, Distance, DistanceMap, DistanceCompare>> as icmp_type;
        icmp_type icmp = new icmp_type(distance, compare);
        typedef mutable<queue <Vertex, VerticesSizeType, icmp_type>> as queue_type;
        queue_type Q = new queue_type(icmp);
        typedef dijkstra.bfs.visitor <GraphT, queue_type, WeightMap, PredecessorMap, DistanceMap, DistanceCombine, DistanceCompare, Vertex, Edge, Distance> as vis_type;
        vis_type bfs_vis = new vis_type(Q, weight, predecessor, distance, combine, compare, zero);
        typedef simple<property_map <Vertex, ColorValue>> as color_map_type;
        color_map_type color = new color_map_type();
        // initialize color map
        foreach (Vertex v in g.vertices()) {
            color[v] = ColorValue.white;
        }
        breadth_first_visit.go
        (g, s, bfs_vis, color, Q);
    }
}
Dijkstra shortest paths

• Next: The need to repeat constraints that are implied by a concept requirement
public class dijkstra_shortest_paths {
    public static void go<
        GraphT, Vertex, Edge, VertexIterator, VerticesSizeType, OutEdgeIterator,
        DegreeSizeType, WeightMap, DistanceMap, Distance,
        PredecessorMap, DistanceCombine, DistanceCompare >
    (GraphT g, Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
     DistanceCombine combine, Distance inf, Distance zero) {
        foreach (Vertex u in g.vertices()) {
            distance [u] = inf;
            predecessor [u] = u;
        }
        distance [s] = zero;
    }
}

typedef indirect cmp < Vertex, Distance, DistanceMap, DistanceCompare > as icmp.type;
icmp.type icmp = new icmp.type(distance, compare);
typedef mutable.queue < Vertex, VerticesSizeType, icmp.type > as queue.type;
queue.type Q = new queue.type(icmp);
typedef dijkstra.bfs.visitor < GraphT, queue.type, WeightMap, PredecessorMap, DistanceMap, DistanceCombine, DistanceCompare, Vertex, Edge, Distance > as vis.type;
vis.type bfs.vis = new vis.type(Q, weight, predecessor, distance, combine, compare, zero);

typedef simple.property.map < Vertex, ColorValue > as color.map.type;
color.map.type color = new color.map.type();

    // initialize color map
    foreach (Vertex v in g.vertices()) {
        color[v] = ColorValue.white;
    }

    breadth_first_visit.go

public class dijkstra_shortest_paths {
    public static void go::<
        GraphT, Vertex, Edge, VertexIterator, VerticesSizeType, OutEdgeIterator,
        DegreeSizeType, WeightMap, DistanceMap, Distance,
        PredecessorMap, DistanceCombine, DistanceCompare>
        (GraphT g, Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
         DistanceCombine combine, Distance inf, Distance zero)
    where GraphT: VertexListAndIncidenceGraph::<
        Vertex,Edge,VertexIterator,VerticesSizeType,OutEdgeIterator,DegreeSizeType>,
    Edge: GraphEdge::<
        Vertex>,
    VertexIterator: IEnumerable::<
        Vertex>,
    OutEdgeIterator: IEnumerable::<
        Edge>,
    WeightMap: ReadablePropertyMap::<
        Vertex,Distance>,
    DistanceMap: ReadWritePropertyMap::<
        Vertex,Distance>,
    PredecessorMap: ReadWritePropertyMap::<
        Vertex,Vertex>,
    DistanceCombine: BinaryFunction::<
        Distance>,
    DistanceCompare: StrictWeakOrdering::<
        Distance> {
        foreach (Vertex u in g.vertices()) {
            distance [u] = inf;
            predecessor [u] = u;
            distance [s] = zero;
        }
        typedef indirect cmp < Vertex, Distance, DistanceMap, DistanceCompare > as icmp.type;
        icmp.type icmp = new icmp.type(distance, compare);
        typedef mutable queue < Vertex, VerticesSizeType, icmp.type > as queue.type;
        queue.type Q = new queue.type(icmp);
        typedef dijkstra_bfs visitor < GraphT, queue.type, WeightMap, PredecessorMap, DistanceMap, DistanceCombine, DistanceCompare, Vertex, Edge, Distance > as vis.type;
        vis.type bfs_vis = new vis.type(Q, weight, predecessor, distance, combine, compare, zero);
        typedef simple property map < Vertex, ColorValue > as color.map.type;
        color.map.type color = new color.map.type();

        // initialize color map
        foreach (Vertex v in g.vertices()) {
            color[v] = ColorValue.white;
        }

        breadth_first_visit.go
    }
}
Dijkstra shortest paths

- Next: Cannot encapsulate associated types into concepts

- The last slide of the sequence repeats the original code as a contrast to the simplified version
public class dijkstra_shortest_paths {
  public static void go<T, Vertex, Edge, VertexIterator, VerticesSizeType, OutEdgeIterator, DegreeSizeType, WeightMap, DistanceMap, Distance, PredecessorMap, DistanceCombine, DistanceCompare>
      (GraphT g, Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
       DistanceCombine combine, Distance inf, Distance zero)
  {
    WeightMap: ReadablePropertyMap< Edge, Distance >,
    DistanceMap: ReadWritePropertyMap< Vertex, Distance >,
    PredecessorMap: ReadWritePropertyMap< Vertex, Vertex >,
    DistanceCombine: BinaryFunction< Distance >,
    DistanceCompare: StrictWeakOrdering< Distance > { 
      foreach (Vertex u in g.vertices()) {
        distance [u] = inf;
        predecessor [u] = u;
      }
      distance [s] = zero;
      typedef indirect cmp< Vertex, Distance, DistanceMap, DistanceCompare > as icmp_type;
      icmp_type icmp = new icmp_type(distance, compare);
      typedef mutable queue< Vertex, VerticesSizeType, icmp_type > as queue_type;
      queue_type Q = new queue_type(icmp);
      typedef dijkstra.bfs.visitor< GraphT, queue_type, WeightMap, PredecessorMap, DistanceMap, DistanceCombine, DistanceCompare, Vertex, Edge, Distance > as vis_type;
      vis_type bfs_vis = new vis_type(g, Q, weight, predecessor, distance, combine, compare, zero);
      typedef simple.property.map< Vertex, ColorValue > as color_map_type;
      color_map_type color = new color_map_type();
      // initialize color map
      foreach (Vertex v in g.vertices()) {
        color[v] = ColorValue.white;
      }
      breadth_first_visit.go
  }
}
public class dijkstra_shortest_paths {
    public static void go<
        GraphT, WeightMap, DistanceMap, Distance,
        PredecessorMap, DistanceCombine, DistanceCompare
        (GraphT g, GraphT.Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
        DistanceCombine combine, Distance inf, Distance zero)
        where GraphT: VertexListAndIncidenceGraph
        
        (GraphT g, GraphT.Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
        DistanceCombine combine, Distance inf, Distance zero)
        where GraphT: VertexListAndIncidenceGraph
        
        (GraphT g, GraphT.Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
        DistanceCombine combine, Distance inf, Distance zero)
        where GraphT: VertexListAndIncidenceGraph
        
        (GraphT g, GraphT.Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
        DistanceCombine combine, Distance inf, Distance zero)
        where GraphT: VertexListAndIncidenceGraph
        
        (GraphT g, GraphT.Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
        DistanceCombine combine, Distance inf, Distance zero)
        where GraphT: VertexListAndIncidenceGraph
        
        (GraphT g, GraphT.Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
        DistanceCombine combine, Distance inf, Distance zero)
        where GraphT: VertexListAndIncidenceGraph
        
        (GraphT g, GraphT.Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
        DistanceCombine combine, Distance inf, Distance zero)
        where GraphT: VertexListAndIncidenceGraph
        
        (GraphT g, GraphT.Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
        DistanceCombine combine, Distance inf, Distance zero)
        where GraphT: VertexListAndIncidenceGraph
        
        (GraphT g, GraphT.Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
        DistanceCombine combine, Distance inf, Distance zero)
        where GraphT: VertexListAndIncidenceGraph
        
        (GraphT g, GraphT.Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
        DistanceCombine combine, Distance inf, Distance zero)
        where GraphT: VertexListAndIncidenceGraph
        
        (GraphT g, GraphT.Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
        DistanceCombine combine, Distance inf, Distance zero)
        where GraphT: VertexListAndIncidenceGraph
        
        (GraphT g, GraphT.Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
        DistanceCombine combine, Distance inf, Distance zero)
        where GraphT: VertexListAndIncidenceGraph
        
        (GraphT g, GraphT.Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
        DistanceCombine combine, Distance inf, Distance zero)
        where GraphT: VertexListAndIncidenceGraph
        
        (GraphT g, GraphT.Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
        DistanceCombine combine, Distance inf, Distance zero)
        where GraphT: VertexListAndIncidenceGraph
        
        (GraphT g, GraphT.Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
        DistanceCombine combine, Distance inf, Distance zero)
        where GraphT: VertexListAndIncidenceGraph
        
        (GraphT g, GraphT.Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
        DistanceCombine combine, Distance inf, Distance zero)
        where GraphT: VertexListAndIncidenceGraph
        
        (GraphT g, GraphT.Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
        DistanceCombine combine, Distance inf, Distance zero)
        where GraphT: VertexListAndIncidenceGraph
        
        (GraphT g, GraphT.Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
        DistanceCombine combine, Distance inf, Distance zero)
        where GraphT: VertexListAndIncidenceGraph
        
        (GraphT g, GraphT.Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
        DistanceCombine combine, Distance inf, Distance zero)
        where GraphT: VertexListAndIncidenceGraph
        
        (GraphT g, GraphT.Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
        DistanceCombine combine, Distance inf, Distance zero)
        where GraphT: VertexListAndIncidenceGraph
        
        (GraphT g, GraphT.Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
        DistanceCombine combine, Distance inf, Distance zero)
        where GraphT: VertexListAndIncidenceGraph
        
        (GraphT g, GraphT.Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
        DistanceCombine combine, Distance inf, Distance zero)
        where GraphT: VertexListAndIncidenceGraph
        
        (GraphT g, GraphT.Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
        DistanceCombine combine, Distance inf, Distanc...
public class dijkstra_shortest_paths {
    public static void go<
        GraphT, Vertex, Edge, VertexIterator, VerticesSizeType, OutEdgeIterator,
        DegreeSizeType, WeightMap, DistanceMap, Distance,
        PredecessorMap, DistanceCombine, DistanceCompare>
    (GraphT g, Vertex s, PredecessorMap predecessor, DistanceMap distance, WeightMap weight, DistanceCompare compare,
     DistanceCombine combine, Distance int, Distance zero)
    {
        // GraphT: VertexListAndIncidenceGraph <Vertex,Edge,VertexIterator,VerticesSizeType,OutEdgeIterator,DegreeSizeType>,
        // Vertex: IVertex
        // Edge: IEdge
        // WeightMap: IWeightMap
        // DistanceMap: IDistanceMap
        // PredecessorMap: IPredecessorMap
        // DistanceCombine: IDistanceCombine
        // DistanceCompare: IDistanceCompare
        //
        // for (IEnumerator<Vertex> ui = g.vertices().GetEnumerator(); ui.MoveNext();)
        // {
        //     Vertex u = ui.Current;
        //     (DistanceMap)distance[u] = inf;
        //     (PredecessorMap)predecessor[u] = u;
        // }
        // (DistanceMap)distance[s] = zero;

       _vertex = ui.Current;
        (DistanceMap)distance[ui] = inf;
        (PredecessorMap)predecessor[ui] = u;
    }
}

indirect_cmp<GraphT, Distance, DistanceMap, DistanceCompare> icmp =
    new indirect_cmp<GraphT, Distance, DistanceMap, DistanceCompare> (distance, compare);

mutable_queue<GraphT, VerticesSizeType, indirect_cmp<GraphT, Distance, DistanceMap, DistanceCompare>> Q
    = new mutable_queue<GraphT, VerticesSizeType, indirect_cmp<GraphT, Distance, DistanceMap, DistanceCompare>> (icmp);

for (IEnumerator<Vertex> ui = g.vertices().GetEnumerator(); ui.MoveNext();)
    {
        Vertex u = ui.Current;
        (DistanceMap)distance[u] = zero;
        
    }

// Initialize color map
for (IEnumerator<Vertex> ui = g.vertices().GetEnumerator(); ui.MoveNext();)
    { }
Summary of language support

- **C++**: Full support, but minimal safety
- **OO languages**: Minimal support, and a lot of safety, but many problems: no retroactive modeling, no direct support for associated types, no direct support for multi-parameter concepts, no type aliases
  - **Java**: Type erasure prevents multi-parameter concepts, current implicit instantiation algorithm prevents emulation of associated types
  - **C#**: (Explicit instantiation), proposed implicit instantiation algorithm seems to have same problem as Java
  - **Eiffel**: Extra problems due to covariance, explicit instantiation, (only one constraint per type argument)
    - BTW, covariance, even safe covariance, does not seem to be important for generic programming
- **ML**: Good support, except for some annoyances; explicit instantiation
- **Haskell**: Very good support (in current implementations), but associated types are not directly supported
Conclusions

- Generic programming is more than `list<T>`
- Flexibility and safety are both important
- Language support:
  - OO subtyping is not a good approximation for modeling
  - ML structures, signatures, and functors are powerful, but somewhat inconvenient to use
  - Type classes provide better (but not perfect) support for generic programming
- Languages (e.g., C#) should be adapted to more directly support generic programming
  - Conjecture: a lot of this can be done without abandoning object-oriented model