Experience with Generic C#

Peter Sestoft (sestoft@dina.kvl.dk)

Royal Veterinary and Agricultural University
and
IT University of Copenhagen
Denmark

References:

Kennedy, Syme: Design and Implementation of Generics for the .NET Common Language Runtime, PLDI 2001.

Syme: ILX: Extending the .NET Common IL for Functional Language Interoperability; section 5. Proceedings of BABEL'01. Electronic Notes in Theoretical Computer Science 59, 1 (2001).

C is unsafe even at simple types; Java isn't

The C programming language has no `boolean` type and will not detect the mistake in this function:

```
double atan2(double y, double x) {
  if (x = 0.0)
    return sign(y) * 3.14159265358979323846 / 2;
  else
    ... atan(y/x) ...
}
```

A Java compiler would report a type error: `if` expects a `boolean`, but `x = 0.0` has type `double`.

But, when we use collection classes, Java provides no compiletime type safety:

```
LinkedList names = new LinkedList();
names.add(new Person("Kristen"));
names.add(new Person("Bjarne"));
names.add(new Integer(1998));          // Wrong, but no compiletime check
names.add(new Person("Anders"));
...
Person p = (Person)names.get(2);       // Cast needed, may fail at runtime
```

The compiletime element type is `Object`, not `Person`.

This is because collection classes must work for all types of elements.

Life with object-based collection classes

Since elements of collections have type `Object`:

- they are dynamically typed: programming errors are discovered only at runtime;
- runtime casts are needed, which slows down the program;
- primitive type values (eg. `int`) must be wrapped (as `Integer`), which takes space and time.

To document non-trivial uses of collections, one may insert comments, ignored by the compiler:

```
Map /* from Integer to Map from String to Integer */ newtrans = new HashMap();
```

Generics can make general code typesafe

With generic collections (parametric polymorphism) one can write instead:

```
IMap<int,IMap<string,int>> newtrans = new HashMap<int,IMap<string,int>>();
```

Advantages:

- the program becomes statically typed, so errors are discovered at compile-time, not in front of the user;
- no runtime casts are needed, so the program is faster;
- primitive type values (eg. `int`) need not be wrapped, so the program is faster and uses less space.

Background: C# (1999)

An object-imperative language with features from Java, C++, Borland Object Pascal, CLOS, Visual Basic.

- Roughly, C# is Java
  - **plus** properties, indexers, user-defined operators, events, `ref` and `out` parameters, ...
  - **plus** value types à la C structs, enumerations, delegates (functions as values), ...
  - **plus** easy integration with Visual Basic, C++, JScript/ECMAScript, Standard ML, ...
  - **plus** conditional compilation, versioning, escape to unsafe (C-style) code, ...
  - **minus** inner classes, `throws` clauses on methods, ...
- C# has been standardized (13 December 2001) as ECMA-334 by Microsoft, Hewlett-Packard, Intel:
  See `http://msdn.microsoft.com/net/ecma/`
- compact 'executables': eg. 5 KB so-called `.exe` files, but not `.exe` files as you know them
- faster program startup than Java
- otherwise comparable to Sun JDK Hotspot 1.3/1.4 performance

Dare Obasanjo has written a good comparison of Java and C#, at
`http://www.prism.gatech.edu/~gte855q/CsharpVsJava.html`

Some C# examples

Hello world in C#:

```
class MyClass {
  static void Main(string[] args) {
    System.Console.WriteLine("Hello, " + args[0]);
  }
}
```

Most of C# is immediately recognizable to a Java programmer.

Some C# specific constructs:

A *property* `Count` of type `int` corresponds to a method `int getCount()`:

```
class LinkedList {
  private int size;

  public int Count {
    get { return size; }                    // Count is a 'get' property
  }
}

LinkedList lst = ... ;
if (lst.Count > 0) ...                      // Using the Count property
```


More C# specific things

An *indexer* with argument `T` and result `U` corresponds to methods `U get(T)` and `void set(T, U)`:

```
class LinkedList {
  private object node(int i) { ... }

  public object this[int i] {               // Indexer: arg int, res object
    get { return node(i).item; }
    set { node(i).item = value; }
  }
}

LinkedList lst = ... ;
if (lst[2] == null)                         // Use the indexer's get
  lst[2] = "N/A";                           // Use the indexer's set
```

The statement `foreach (T x in C) S` uses an underlying enumerator (iterator), roughly:

```
IEnumerator enm = C.GetEnumerator();
while (enm.MoveNext()) {
  T x = (T)enm.Current;
  S;
}
```


C# example: Object-based quicksort of `arr[a..b]`

```
public interface IComparable {
  int CompareTo(object that);
}

private static void qsort(IComparable[] arr, int a, int b) {
  if (a < b) {
    int i = a, j = b;
    IComparable x = arr[(i+j) / 2];
    do {
      while (arr[i].CompareTo(x) < 0) i++;
      while (x.CompareTo(arr[j]) < 0) j--;
      if (i <= j) {
        swap(arr, i, j);
        i++; j--;
      }
    } while (i <= j);
    qsort(arr, a, j);
    qsort(arr, i, b);
  }
}
```

The ordering is fixed, determined by the method `CompareTo` belonging to each array element.

Generic C# (2001)

- Generic types, such as `LinkedList<T>`
- Generic methods, such as `Quicksort<T>(T[] arr)`
- Instantiation at primitive types (eg. `int`) and reference types (eg. `string`)
- Exact runtime types, cast, and instance check (`Equals` in `LinkedList<T>`)
- Constraints on type parameters; a constraint may involve the type parameter itself
- No covariance in the type parameters of a generic type
- Type parameter `T` cannot be used in `new T()` or `T.m(...)`
- A type cannot implement a generic interface at more than one instance:
  `class Matrix : IEnumerable<int>, IEnumerable<int[]>`
- Polymorphic recursion possible: `void m<T>(T x) { ... m<IList<T>>(...) ... }`

Generic C# example: quicksort of `arr[a..b]`

```
public interface IComparer<T> {
  int Compare(T v1, T v2);
}

private static void qsort<T>(T[] arr, IComparer<T> cmp, int a, int b) {
  if (a < b) {
    int i = a, j = b;
    T x = arr[(i+j) / 2];
    do {
      while (cmp.Compare(arr[i], x) < 0) i++;
      while (cmp.Compare(x, arr[j]) < 0) j--;
      if (i <= j) {
        swap<T>(arr, i, j);
        i++; j--;
      }
    } while (i <= j);
    qsort<T>(arr, cmp, a, j);
    qsort<T>(arr, cmp, i, b);
  }
}
```

The ordering is determined by a separate method `cmp.Compare`.

More flexible than the `IComparable` approach; people may be sorted by name, age, birthday (Jan–Dec).

Generic C# example: doubly linked list implementation

```
public class LinkedList<T> : IList<T> {
  Node<T> first, last;          // Invariant: first==null iff last==null

  private class Node<T> {       // Nested class
    public Node<T> prev, next;
    public T item;
  }

  public T Get(int n) { ... }

  public bool Add(T item) { ... }

  public override bool Equals(object that) {    // Inherited from object
    if (that is IList<T>) {                     // Exact instanceof check
      ...
  }
}
```

Method `Get(int)` returns a `T`, not an `object`.

Generic doubly linked list, example use

```
LinkedList<Person> names = new LinkedList<Person>();
names.Add(new Person("Kristen"));
names.Add(new Person("Bjarne"));
names.Add(new Integer(1998));           // Wrong, compiletime error
names.Add(new Person("Anders"));
...
Person p = names.Get(2);                // No cast needed at runtime
```


Type parameter constraints

A Printable has a Print method; a `List<T>` is Printable when `T` is:

```
interface Printable {
  void Print(TextWriter fs);
}

class List<T : Printable> : Printable {
  public void Print(TextWriter fs) {
    foreach (T x in this)
      x.Print(fs);
  }
}
```

An `IGComparable<T>` may be compared to values of type `T`:

```
public interface IGComparable<T> {
  int CompareTo(T that);
}
```

Values of type `T` may be sorted if they implement `IGComparable<T>`; note `T` in constraint:

```
private static void qsort<T : IGComparable<T>>(T[] arr, int a, int b) {
  ...
  while (arr[i].CompareTo(x) < 0) i++;              // Typesafe
  ...
}
```

Implementation of Generic C#: The Generic Common Language Runtime

Compilation to Generic Intermediate Language

- CIL is the intermediate language of CLR, similar to Java bytecode but less C# or Java specific.
- Generic CIL is CIL with explicit type parameters and validation of generic code.
- Generic classes and methods are typechecked at declaration, not at use (as in C++)
- After typecheck, Generic C# is compiled rather straightforwardly to Generic CIL.

Runtime specialization or sharing of generic code

- Specialized instances of a generic class C<T> are created at runtime for relevant argument types T.
- Type instances are shared among reference types T, avoiding most of the code bloat of C++.
- Type instances are not shared among value types T, giving compact data representation and better speed.
- Type instances are created lazily (this permits polymorphic recursion).
- The vtable for a type instance contains the exact actual argument types.
- These argument types are used in runtime type casts, and instance checks.

Efficiency benefit of generics: quicksort

Description	General	Typesafe	Generics	Ints	Strings
Object-based, interface IComparable	yes	no	no	4.99	3.18
Object-based, class OrderedInt	yes	no	no	3.08	2.58
Generic with untyped CompareTo	yes	no	yes	3.15	2.57
Generic with typed CompareTo	yes	yes	yes	2.45	2.54
Generic with Compare method	yes	yes	yes	1.14	2.19
Generic with Compare delegate	yes	yes	yes	1.91	2.83
Hand-specialized, inline <	no	yes	no	0.47	2.10
Hand-specialized with Compare method	no	yes	no	1.06	2.19

Random ints (1.000.000) or strings (200.000); average time/s of 20 runs; 1 GHz P-III; Windows XP; Generic CLR.

- Generics is the only way to have generality, type safety, and efficiency.
- The only overhead in generics (1.14 vs 0.47) is due to the passing of the Compare method (generality).
- The generics win is clearly larger for the value type `int` than for the reference type `string`.
- The current implementation of delegates (functions as values) is poor.

Fragments of a Java-style generic collection library (the standard .NET collection library is deficient)

```
public interface IEnumerator<T> {       // Iterator (stream) over type T
  T Current { get; }
  bool MoveNext();
}

public interface IEnumerable<T> {       // Has an iterator over type T
  IEnumerator<T> GetEnumerator();       // Supports the foreach statement
}

public interface ICollection<T> : IEnumerable<T> { ... }

public interface ISet<T> : ICollection<T> {
  bool Add(T item);
  bool Contains(T item);
}

public class OTreeSet<T : IComparable> : ISet<T> { ... }

public interface IMap<K,V> : ICollection<MapEntry<K,V>> {
  bool Add(K key, V val);
  V this[K key] { get; set; }
  bool Contains(K key);
}

public class OTreeMap<K : IComparable, V> : IMap<K,V> { ... }
```


GC# example: Create a sorted index of word occurrences in a text file

```
TreeMap<string, TreeSet<int>> index = new OTreeMap<string, TreeSet<int>>();
Regex delim = new Regex("[ \\t‘~!@#$%^&*()_+=;’:\\[\\]\\{\\}<>\\\\\\|\",./?-]+");
TextReader rd = new StreamReader(filename);
int lineno = 0;
string line;
while (null != (line = rd.ReadLine())) {
  lineno++;
  string[] res = delim.Split(line);
  foreach (string s in res)
    if (s != "") {
      if (!index.Contains(s))
        index[s] = new OTreeSet();
      index[s].Add(lineno);
    }
}
rd.Close();
foreach (MapEntry<string, TreeSet<int>> wordlist in index) {
  Console.Write("{0}: ", wordlist.Key);
  foreach (int ln in wordlist.Value)
    Console.Write(ln + " ");
  Console.WriteLine();
}
```

This took most of chapter 13 in Welsh and Elder's Pascal textbook (1980).

Java requires casts and while loops. C# lacks the TreeSet and TreeMap collections.

Sample output

```
...
complicate: 784
conclude: 577 587
conclusion: 165
conflict: 723
considerably: 262 775
constraint: 326 327 581 582
constraints: 446 548 570 572 634 654 729 746 748
constructed: 379 383
constructor: 467 690
constructors: 452
containing: 382
contains: 300 304 306 315
Contains: 427
continuation: 745
convenient: 784
conversion: 325 628 765
...
```


So Generic C# is some kind of glorified Generic Java?

Yes: Generic C# is quite similar to Generic Java, just as C# is quite similar to Java.

And **No:** Generic C# is quite different:

- Implemented by specialization, not type erasure, by the Generic Common Language Runtime.

  Hence considerably more efficient than the current Generic Java proposal.
- The Generic Common Language Runtime will be out there, on millions of machines.
- Generic C# has several desirable features missing from the Generic Java proposal:

  ... it would be a good thing for the Java programming language to add *generic types* and to let the user define *overloaded operators*. [...] What is more, I would add a kind of *class of light weight* ...

  Guy Steele, keynote address at OOPSLA'98; my italics

  GJ is an excellent design and implementation for adding generic types [...] I would hope to see it compatibly extended to *carry run-time type parameter information* ...

  Guy Steele, quoted from e-mail to Phil Wadler; my italics

Comparison to Generic Java (1998)

Bracha, Odersky, Stoutamire, Wadler (OOPSLA 1998), and Java Specification Request 14, August 2001.

Generic classes, interfaces and methods as in Generic C#, and static polymorphic type check.

More types are *inferred* at method calls than in current Generic C#, hence less verbose (but unsound?)

Type parameters are seen also by static members, nested classes, and inner classes.

Instantiation of type parameters only at reference types, not `int`, `double`, ...

Implementation by type erasure: type parameters are replaced by `Object`.

Advantage: code runs on standard Java Virtual Machines.

Disadvantages:

- No efficiency or space gain at primitive types.
- Type parameters cannot be used at runtime.
- Polymorphic array creation `new T[10]` is permitted but not typesafe (compiletime warning), and `(e instanceof T)` is impossible; see `Equals` in `LinkedList<T>`.

Generic Java will probably be Java 2 version 1.5 (in 2003 or 2004?).

Generics are available in an extension of Sun's `javac` compiler, since March 21, 2002.

Java and C# array assignment requires runtime type checks

```
void m(A[] arr, A x) {
  arr[0] = x;                          // Runtime check needed
}
```

Why? Assume `B` is a subclass of `A`, then call

```
B[] arr = new A[10];
m(arr, new A());
... here arr[0] would contain an object of class A, not good ...
```

Lack of exact runtime types make runtime type check impossible

```
void m<T>(T[] arr, T x) {
  arr[0] = x;                          // Runtime check needed
}
```

But the check requires the exact type of `X` at runtime.

OK in Generic C#; not in Generic Java, because of implementation by type erasure.

Other proposals for genericity in Java

PolyJ (Myers, Bank, Liskov; POPL 1997)

Generic types can be instantiated at reference types and primitive types (and array types?).

Type parameters can be constrained using `where`-clauses (from CLU) instead of interfaces and classes.

Implementation requires an extended JVM to handle primitive types, and for efficiency.

Little chance that this will become reality.

NextGen (Cartwright, Steele; OOPSLA 1998)

A type parameter T can be used anywhere a type can, including at `new T()`, not possible in GC#.

Static members see the type parameters; there is a copy of each static member at each type instantiation.

Only classes and interfaces, not base types, can be substituted for type parameters.

Implemented by type erasure, wrapper classes and interfaces, and code snippets; runs on standard JDK 1.2 JVM.

Little chance that this will become reality.

Comparison to C++ templates (1990)

- Templates are checked at instantiation, not declaration. Hence weak compiletime typing of generic libraries.
- Generic libraries can be distributed only in source form.
- Templates produce highly efficient instantiations at the cost of serious code bloat.
- C++ does not permit constraints (bounds) on type parameters.
- Templates can be instantiated at types and *values*, and are more expressive than Java and C# generics (Veldhuizen: Techniques for Scientific C++, 2000).

Comparison to Eiffel

Generic C# seems to have all the desirable features of Eiffel generics (and lack the undesirable ones).

Comparison to ML (1979)

ML was the first language to have parametric polymorphic types.

Types are *inferred* at compiletime, and are mostly implicit; hence much less verbose than Generic C# and Java.

Most ML implementations represent values of primitive type and reference type the same way.

Some compilers use type information to avoid boxing of (e.g.) floating-point numbers, for speed.

Comparison to Cyclone (Morrisett et al, 2000)

Goal: C's efficiency and close control of memory, but safely. A simple language with a rich type system.

- Three pointer types: nullable (`*`, null checked), fat (`?`, bounds checked), never-null (`@`, no runtime check).
- Statically typed, parametric polymorphism (over word-size types):

```
void swap('a@ x, 'a@ y) { // x and y are never-null pointers to 'a
  let tmp = *x; *x = *y; *y = tmp;
}

int main() {
  int a[] = { for i<10 : (int)i };      // array of 10 ints
  swap(&a[2], &a[7]);
  int@ ap[] = { for i<10 : &a[i] };     // array of 10 int pointers
  swap(&ap[3], &ap[4]);
}
```

- No classes and objects, but structs, structural subtyping, unification-based type inference.
- Allocation in *regions* à la Tofte/Talpin; values cannot be `free`'d individually; the heap is garbage collected.
- Implemented by compilation to C; no types at runtime, no code specialization.
- Leaving types to be inferred can change behaviour (because of implicit conversions). A warning to GC#.
- The type system is rich, with error messages to go with it:

```
actual argument has type int @'main but formal has type int @'main @%(77)::R
```


Generics by virtual types (Beta)

A generic class `class C<T> { ... }` is not itself a class, but a function from types to classes.

Object-oriented hardliners want to rectify this with *virtual types* instead of type parameters.

A virtual type can be specified in a class and 'further-bound' in subclasses:

```
interface I {
  typedef ST as Object;
  void m(ST st) { ... }
}

class F implements I {
  typedef ST as String;                        // Further-binding
  void m(ST st) { ... st.length() ... }
}
```

But it requires runtime type checks. Assume:

```
I obs;
obs = new F();
```

Since `obs` has type `I` and `I.ST` is `Object`, this should typecheck at compiletime:

```
obs.m(new Integer());
```

But now clearly `st.length()` will fail, as `Integer` has no `length` method.

So no compiletime safety. Or I'm missing the point, quite probably.

Evaluation of Generic C#: good

- No covariance of parametrized types: simplicity and safety.
- Explicit types provide compiler checkable documentation; found errors in existing carefully commented code.
- Programs can be more efficient because object wrappers and runtime casts are not needed.
- The design appeals to programmers used to Java and SML; there are few surprises.

Evaluation of Generic C#: some pitfalls

- No co-variance: `OTreeMap<int,OTreeSet<int>>` not a subtype of `IMap<int,ISet<int>>`
- Types must be explicit and can get rather unwieldy (but abbreviations and inference have been proposed):

```
static IMap<int, IMap<string, int>>
  rename(IMap<HashSet<int>, int> renamer,
         IMap<HashSet<int>, IMap<string, HashSet<int>>> trans) { ... }
```

- C# automatically converts value types (`int`) to reference types (`object`), so `42.ToString()` is legal.

  So mixing generic and object-based code may introduce hard-to-predict inefficiencies.

  Storing an `int` in an object-based `TreeSet` requires a single initial conversion.

  But using object-based comparisons inside `TreeSet<int>` requires conversion at every comparison . . .

Will Generic C# tie us irreversibly to Microsoft?

Yes: Since January 2002, free command-line compiler and runtime for C# is available for Windows 2000 and XP.

On November 8, 2002 Microsoft officially announced that they will release Generic C# (but not when):

`http://www.microsoft.com/presspass/press/2002/Nov02/11-08OOPSLAPR.asp`

But **No:** In the long run CLR and (Generic) C# will be available elsewhere:

- There are open source projects to implement CLR and C#:

  The Mono CLR and C# implementation (www.go-mono.com) for Linux is developing rapidly.

  The Mono developers are aware of the Generic CLR work and say they will consider generics as well.
- Microsoft has a Shared Source CLR implementation ('Rotor') for FreeBSD and Windows XP:

  `http://www.microsoft.com/partner/products/microsoftnet/SharedSourceCsharpCLIFAQ.asp`

  Since September 4, 2002 Generic C# is available as a patch to Shared Source CLR — incl. my examples.

  So generics should be in the next Shared Source CLR release (2003?)

Which is best for teaching?

- Java is simpler (except for inner classes), and thus possibly a better teaching language.
- The similarity of Java and C# makes the choice quite insignificant; easy to go from one to the other.
- The discipline of compile-time checkable types should be taught, in Generic Java or Generic C#.

Conclusion

Generic C# provides some expressiveness and compiletime type safety absent in most mainstream languages.

Generic C# is a well-designed language with a high-tech implementation.

Kennedy and Syme had an opportunity to push some academic work into practice, and exploited it very well.

Infinite code and type specialization possible (in contrived examples)

Unbounded code specialization for structs; `m1<int>(n, 42)` creates a struct type containing $n+1$ integers:

```
struct Foo<T> {
  int i;                        // take up some space
  T t1;                         // and some more
}

static void m1<U>(int n, U s) {
  if (n > 0)
    m1<Foo<U>>(n-1, new Foo<U>());
}
```

Bounded code but unboundedly many types; evaluating `new C<int>(n)` creates $n+1$ type instances of `C`:

```
class C<T> {
  public C<C<T>> c;

  public C(int n) {
    if (n > 0)
      c = new C<C<T>>(n-1);
  }
}
```

Type instances are created lazily, so unboundedly many types requires unboundedly many objects.