Experience with Generic C#

Peter Sestoft (sestoft@dina.kvl.dk)
Royal Veterinary and Agricultural University
and
IT University of Copenhagen
Denmark

References:

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 compile-time type safety:
LinkedList names = new LinkedList();
names.add(new Person("Kristen"));
names.add(new Person("Bjarne"));
names.add(new Integer(1998)); // Wrong, but no compile-time check
names.add(new Person("Anders"));
...
Person p = (Person) names.get(2); // Cast needed, may fail at runtime

The compile-time 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;
primitivetype values (e.g. 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 type-safe.

The following code is general:

```csharp
public class Person {
  public String name;
}
public class Integer {
  public int value;
}
```

```csharp
public class MyMap<key, value> {
  public Dictionary<key, value> map;
}
```

```csharp
MyMap<Person, Integer> newtrans = new MyMap<Person, Integer>();
```

Advantages:
the program becomes statically typed, so errors are discovered at compile-time, not in front of the user;
noruntime casts are needed, so the program is faster;
primitivetype values (e.g. 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 valuetype `ala 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': e.g. 5 KB so-called .exe files, but not .exe files as you know them
Much faster program startup than otherwise comparable to SunJDK Hotspot 1.3 performance.
Dare Obasanjo has written a good comparison of Java and C#, at http://www.prism.gatech.edu/~gte855q/CsharpVsJava.html

Experience with Generic C#
Some C# examples

Hello 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():
C# example: quicksort of arr[a..b]

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

```csharp
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: `Compare`.

```
int Compare(T that)
{ ... }
```

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

```
public interface IComparer<T>
{
    int Compare(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++;
    ... }
```

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

Generic C# example: doubly linked list implementation

```csharp
public class LinkedList<T> : IList<T>
{
    Node<T> first, last; // Invariant: first == null if 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 instance of check
        {
            ... }
    }
    Method Get(int) returns a T, not an object.
```

Generic doubly linked list, example use

```csharp
LinkedList<Person> names = new LinkedList<Person>();
names.Add(new Person("Kristen"));
names.Add(new Person("Bjarne"));
names.Add(new Person("Anders"));
... Person p = names[2]; // No cast needed
```

Type parameter constraints

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

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

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

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:

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

```csharp
... Method Get(int) returns a T, not an object.
```
Implementation of Generic C#: The Generic Common Language Runtime

Compilation to Generic Intermediate Language

IList is the intermediate language of CLR, similar to Java bytecode but less C# or Java-specific.

Generic IL is IL with explicit type parameters and validation of generic code.

Generic classes and methods are type checked at declaration, not at use (as in C++).

After type check, Generic C# is compiled rather straightforwardly to Generic IL.

Runtimespecialization 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 typecasts and instance checks.

Efficiency benefit of generics: quicksort

Description

General Typesafe Generics

Ints Strings

Object-based, interface IComparable

yes no

4.99 3.18

Object-based, class OrderedInt

yes no

3.08 2.58

Generic with untyped CompareTo

yes no

3.15 2.57

Generic with typed CompareTo

yes yes

2.45 2.54

Generic with Compare method

yes yes

1.14 2.19

Generic with Compare delegate

yes yes

1.91 2.83

Hand-specialized, inline

no yes

0.47 2.10

Hand-specialized with Compare method

no yes

1.06 2.19

Random ints (1.000,000) or strings (200,000); average time/sof 20 runs; 1GHz P-III; Windows XP; Generic CLR.

Generics is the only way to have generality, typesafety, and efficiency.

The only overheading generics (1.14 vs 0.47) is due to the passing of the Compare method (generality).

The generics win is clearly larger for the valuetype int than for therefertypetype string.

The current implementation of delegates (functions as values) is poor.

Fragmentsofa 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
    bool Contains (IEnumerator<T> other);
}

public interface ICollection<T> : IEnumerable<T>
{
    // Has an iterator over type T
    IEnumerator<T> GetEnumerator();
    bool Contains (IEnumerator<T> other);
    int Count { get; } // not implemented in .NET Framework
}
So GenericC# is some kind of glorified Generic Java?

Yes:

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

And No:

GenericC# 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.

The implementation seems better engineered than Sun’s Java implementations (mostly).

GenericC# 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 lightweight

...

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 parameter information

...
ComparisontoC++templates (1990)

Templates are checked at instantiation, not declaration. Hence weak compile-time 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.

Templates can be instantiated at types and values, and are more expressive than Java and C# generics (Veldhuizen: Techniques for Scientific C++, 2000).

ComparisontoEiffel

Generic C# seems to have all the desirable features of Eiffel generics and lacks some undesirable ones.

ComparisontoML (1979)

ML was the first language to have parametric polymorphic types. Types are inferred at compile time, and are mostly implicit; hence much less verbose than Generic C# and Java.

Most ML implementations represent values of primitive types and reference types the same way. Some compilers use type information to avoid boxing of (e.g.) floating-point numbers, for speed.

Evaluation of Generics: some problems

The design appears to be powerful enough to do all and still: new problems.

Programs that use generic methods often create and manage types as needed.

Explicit casts provide better documentation than existing equality-compared code.

No language support for safety.

Evaluation of Generics: good

No co-variance: 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; few pitfalls.

Evaluation of Generics: some pitfalls

No co-variance: OTreeMap<int, OTreeSet<int>> is not a subtypeof 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) {...}

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

Some combination of performance to boxing of (g) floating-point numbers, for speed:

Or I'm missing the point, quite probably.

Generics by virtual types (Beta)

A generic class

\[
\text{class C<T>}{\ldots}\]

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:

\[
\text{interface I}{\ldots}\]

\[
\text{typedef ST as Object;}{\ldots}\]

\[
\text{void m(ST st)\{\ldots\}}{\ldots}\]

\[
\text{class F implements I}{\ldots}\]

\[
\text{typedef ST as string;}{\ldots}\]

\[
\text{void m(ST st)\{\ldots\} st.length()\ldots}\]

But it requires run-time checks. Assume:

\[
\text{I obs;}{\ldots}\]

\[
\text{obs=new F();}{\ldots}\]

Since \(\text{obs}\) has type \(\text{I}\) and \(\text{I}\).ST is \text{Object}, this should type check at compile time:

\[
\text{obs.m(new Integer());}{\ldots}\]

But now clearly \(\text{st.length()}\) will fail, as \(\text{Integer}\) has no \text{length} method.

So compile-time safety. Or I'm missing the point, quite probably.
Will C# tie us all to Microsoft Windows?

Yes:
Currently C# is available only for Windows 2000 and XP (after lengthy installation of .NET beta 2). (Runtime and command-line compilers will be free.)
Currently Generic C# is not available at all outside of Microsoft.

No:
In the long run, CLR and C# will be available elsewhere:
There are open-source projects to implement CLR and C#:
- Mono (www.go-mono.com), DotGNU (www.dotgnu.org), and Southern Storm.
The Mono C# compiler is developing rapidly, more or less working for Linux as of December 2001.
The Mono people are aware of the Generic CLR work and seem to consider generics as well.
- Microsoft has a "shared source" project called Rotor for CLR on FreeBSD and Windows XP:
  http://www.microsoft.com/partner/products/microsoftnet/SharedSourceCsharpCLIFAQ.asp
  Or search for "rotor freebsd microsoft" on Google and you'll get there.
Generics maybe in first shared source release (2002)?

(Aside: CLR could become a universal computer language (UNCOL) — sought since 1958.)

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 compile-time type safety absent in most mainstream languages.

Generic C# is a well-designed language with a high-tech implementation.
Generic C# provides some expressiveness and compile-time type safety absent in most mainstream languages.

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

Commonalities:
- Microsoft's CLR is the new standard for managed code.
- IBM, Sun, HP, DEC, HP, and others have their own versions.
- Java is the implementation for the Generic Java work to be done.

The Mono and Southern Storm projects are the most promising of the open-source implementations.

Why (why-Oh-no-man-computer-world-now!)?
- .NET is a shared source project opened up to FreeBSD and Windows XP.
- The Mono people are aware of the Generic CLR work and seem to consider generics as well.
- More (why-Oh-no-man-computer-world-now!)?
- There are some problems to implement the CLR and C#.

But no to the Longhorn CLR and C# will be available elsewhere.

Currently the Generic CLR is just an academic effort outside of Microsoft.

Will (Windows) CE be just another Windows?