Concepts in Programming Languages
Lecture 8: Java and C#

Andrew Kennedy
Microsoft Research Cambridge

How did we get here?

• This lecture: design of Java and C#
  – Origins
  – Evolution
  – Influences
• Focus on C#
  – I know Java
  – I know C#
  – Lots of interesting features in upcoming C# 3.0

Java: origins

• Sun’s “Oak” project (1991): a language for programming consumer
  electronic devices
• Didn’t take off, but formed basis for Java (1995), a language for the web
  – Java Virtual Machine was integrated in Sun’s HotJava web browser; Netscape
    and Microsoft followed suit
  – Original aim was “applets” running inside the browser
  – Also “servlets” processing web queries on server
  – Client-side Java programming for standalone apps
  – Also in devices (particularly smartcards, mobile phones) closer to original
    vision for Oak
• Ironically, “applets” failed, but look at the Google Web Toolkit: compiling
  Java to Javascript to run in the browser!

Java: design goals

• Simplicity
  – perhaps a reaction against C++
  – though even Java 1.0 had complex features: e.g. overloading
    resolution, multiple class loaders
• Safety and security
  – strongly typed (mostly static)
  – automatic memory management (= no pointer errors)
  – code access security by “stack inspection”
• Portability
  – compiled to bytecode, executed by Java Virtual Machine
• Object-orientation
  – simple model of implementation inheritance for classes
  – multiple interface inheritance

Java: impact

• Many ideas were not new
  – e.g. Pascal p-code, Smalltalk virtual machine
  – e.g. Modula-3 had shown practicality of type-safe
    language with garbage collection
• But Java has had a big impact
  – popularized automatic memory management
  – encouraged novel compilation technology (just-in-time compilation, dynamic re-compilation, runtime
    specialization, etc)
  – much studied by researchers (type systems, semantics, static analysis, concurrency)
C#: origins

- 1999: .NET, a new framework for application development
  - a Common Language Runtime + variety of languages (C#, C++, Visual Basic, Eiffel, now Python and others)
  - initial focus: server-side web programming, web services
  - more generally, client-side application development
  - original hope was that many components of Vista would be written using managed code (C#). Unfortunately, not realised.
  - but for another (research) OS – “Singularity” – even device drivers are coded in C#. Type safety of C# plays key role ensuring integrity of “software isolated processes”

C# vs Java: object model

- Object model
  - Similar core: primitive types (int, float, etc), single-inheritance classes + multiple interfaces, covariant arrays
  - Better versioning properties e.g. add same-name method to classes or new interface without accidental override
  - Built-in notion of “boxing” for converting primitive values to heap-allocated objects
  - Lightweight struct à la C++; can implement interfaces just like classes

C# vs Java: parameter passing

- Default is call-by-value
- Explicit annotation for call-by-reference
  - in/out e.g.
  - out e.g.
  - Semantics is deliberately under-specified: typically, local calls are call-by-reference, remote calls are copy-in, copy-out

C# vs Java: first-class functions

- Observation: object-oriented programming is higher-order programming
  - Formally, can translate functions (e.g. from λ-calculus) into objects, representing free variables in fields of an object, and the body of a by an “Apply” method. e.g.

Interlude: type safety


Bertrand Meyer, on unsoundness of Eiffel:
“Eiffel users universally report that they almost never run into such problems in real software development.”
Ten years later: Java

Java is not type-safe

Java is Type Safe — Definitively

Java is Type Safe — Probably

Java evolution: generics

- Java 1.5 has support for parameterized types and polymorphic methods ("generics")

```java
class Stack<T> {
    T[] items;
    int nitems;

    T Pop() {
        if (nitems == 0) { ... } else {
            nitems--;
            return items[nitems];
        }
    }
}
class Array {
    static <T> void Sort(T[] arr) { ... }
}
```

- Compile-time type error!

- Type parameter to class
- Type parameter to method

C# evolution: anonymous methods

- C# 2.0 introduced a lightweight mechanism for first-class functions: the ability to create a delegate object from in-line code, much like a λ-abstraction.

```csharp
delegate bool Predicate(int x);
bool Exists(List<int> list, Predicate p) {
    foreach (int i in list) {
        if (p(i)) return true;
    }
    return false;
}
bool ExistsPrimeAbove(List<int> list, int limit) {
    return Exists(list, delegate (int i) { return i > limit; });
}
```

- Why? Clue: r-values vs l-values. Arguably the right decision for an imperative language:

```csharp
int Func(int x) {
    return x + 1;
}
```

Anonymous methods puzzler

```csharp
delegate bool Predicate(int x);
bool Exists(List<int> list, Predicate p) {
    foreach (int i in list) {
        if (p(i)) return true;
    }
    return false;
}
bool ExistsPrimeAbove(List<int> list, int limit) {
    return Exists(list, delegate (int i) { return i > limit; });
}
```

- Guess the output

```csharp
int Func(int x) {
    return x + 1;
}
```

- Why? Clue: r-values vs l-values. Arguably the right decision for an imperative language:

```csharp
static void While(Predicate condition, Action action) {
    int i = 0;
    while (condition(i)) {
        action(i);
        i++;
    }
}
```

C# vs Java: static typing

- Type safety is now recognised as crucial – a type loophole is a potential security loophole if "untrusted" code is downloaded from the web
- Java’s original type system was simple but limited – e.g. no support for parametric polymorphism
- Escape by downcast, safety ensured by runtime type-check
- Covariant arrays, made type-safe by runtime type-check on update

```csharp
const string s = "abc";
```

```java
String s = "abc";
```

- Java evolution: generics

- Java 1.5 has support for parameterized types and polymorphic methods ("generics")

```java
class Stack<T> {
    T[] items;
    int nitems;

    T Pop() {
        if (nitems == 0) { ... } else {
            nitems--;
            return items[nitems];
        }
    }
}
class Array {
    static <T> void Sort(T[] arr) { ... }
}
```

- Compile-time type error!

- Type parameter to class
- Type parameter to method

C# evolution: generics

- C# 2.0 introduced its own design for generics
- Improves on Java model – Value type instantiations, e.g. List<int>
- No odd restrictions (e.g. ?[T] illegal in Java)
- Types preserved at runtime (e.g. List<string> x really checks that x is a List of strings; in Java it just checks that x is a List
- Better performance due to native support in runtime

- In the meantime, Java introduced its own novelty: "wildcard" types, providing a kind of variance/existential ability. Search the web for opinions on this feature!
C# 1.0 foreach

- Like Java, C# has standard interfaces for "iterators" and "iterable" collections:
  ```csharp
  interface IEnumerable<T> { IEnumerator<T> GetEnumerator(); }
  interface IEnumerator<T> { bool MoveNext(); T Current { get; } }
  ```

- The foreach construct makes it easy to write consumer code.

- Producer code – implementing the IEnumerable and IEnumerator interfaces – is trickier, as the programmer must carefully maintain iterator state when coding up Current and MoveNext methods.

```
foreach (i in List<int>) { sum += i; }
```

C# 2.0 iterators

- C# 2.0 introduces iterators (similar to generators in the Icon programming language), easing task of implementing producers e.g.

```
IEnumerable<int> UpAndDown(int bottom, int top) {
  for (int i = bottom; i < top; i++) { yield return i; }
  for (int i = top; i > bottom; i--) { yield return i; }
}
```

- Iterators can mimic functional-style streams. They can be infinite:
  ```
  static IEnumerable<int> Evens() {
    for (int i = 0; true; i+=2) { yield return i; }
  }
  ```

```
foreach(T x in xs) { if (n>0) n--; else yield return x; }
```

Java generics: implementation

- Two of the design goals for Java generics were
  - no change to the bytecode or JVMs
  - backward compatibility for collection libraries: retrofit generic types onto non-generic library code

- This drove implementation technique of type erasure and is the reason for the odd restrictions. Essentially, generics is "compiled away" e.g.

```
Stack<String> st;
st.Push("abc");
string s = st.Pop();
```

```
Stack st;
st.Push((object)"abc");
string s = (string)st.Pop();
```

C# generics: implementation

- For C#, we were able to change the bytecode and runtime (virtual machine)
- Prototype by Don Syme, Andrew Kennedy and Claudio Russo, code transferred to product for .NET 2.0
- Example of bytecode:
  ```
  static void Swap<T>(ref T x, ref T y) 
  { 
    T tmp = x; 
    x = y; 
    y = tmp; 
  }
  ```

Generics: implementation, as was

Two main techniques:
- **Specialize** code for each instantiation
  - C++ templates, Milton & SML.NET monomorphization
  - good performance 😊
  - code bloat 😞

- **Share** code for all instantiations
  - Either use a single representation for all types (ML, Haskell)
  - Or restrict instantiations to "pointer" types (Java)
  - no code bloat 😊
  - poor performance 😞 (extra boxing operations required on primitive values)

C# generics: implementation

- Runtime does "just-in-time code specialization" but shares representation and code where possible
  - resulting performance almost as good as hand-specialized code

- Rule:
  - share field layout and code if type arguments have same representation

- Examples:
  - Representation and code for methods in Set<string> can be also be used for Set<object> (string and object are both 32-bit pointers)
  - Representation and code for Set<long> is different from Set<int> (int uses 32 bits, long uses 64 bits)
C# generics: implementation

- We wanted to support
  \[
  \begin{align*}
  \text{if } & (x \text{ is } \text{Set\text{<string>}}) \{ \ldots \} \\
  \text{else if } & (x \text{ is } \text{Set\text{<Component>}}) \{ \ldots \}
  \end{align*}
  \]
- But representation and code is shared between compatible instantiations e.g. \text{Set\text{<string>}} and \text{Set\text{<Component>}}
- So there was a conflict to resolve...
  \[\ldots\]
- and we didn't want to add lots of overhead to languages targeting .NET that don’t need run-time types (ML, Haskell)
- Solution was to maintain distinct virtual dispatch tables (so-called v-tables) for each instantiation
  - v-table slots point to shared code
  - cache runtime type information in extra slots in table

C# evolution: LINQ

- Focus of upcoming version 3.0 is Language INtegrated Query
- Slick integration of SQL-like queries into C# requires additional language features, useful in their own right
  - lambdas
  - type inference
  - meta-programming
  - anonymous types
  - extension methods
- We'll take a quick look at the first two.

Lambda expressions

- C# 2.0 anonymous methods are just a little too heavy compared with lambdas in Haskell or ML: compare
  \[
  \begin{align*}
  \text{delegate } & (\text{x, y}) \{ \text{return } x*x + y*y; \} \\
  (\text{x, y}) & \mapsto x*x + y*y
  \end{align*}
  \]
- C# 3.0 introduces \textit{lambda expressions} with a lighter syntax, inference (sometimes) of argument types, and expression bodies:
  \[
  (\text{x, y}) \mapsto x*x + y*y
  \]
- Language specification simply defines lambdas by translation to anonymous methods.

Type inference

- Introduction of generics in C# 2.0, and absence of type aliases, leads to typeful programs!
  \[
  \begin{align*}
  \text{Dict\text{<string,Func\text{<int,Set\text{<int>>>}}} & d = \text{new Dict\text{<string,Func\text{<int,Set\text{<int>>>}}}();} \\
  \text{Func\text{<int,int,int>} & f = \text{delegate } (\text{x, y}) \{ \text{return } x*x + y*y; \};}
  \end{align*}
  \]
- C# 3.0 supports a modicum of type inference for local variables and lambda arguments:
  \[
  \begin{align*}
  \text{var} & d = \text{new Dict\text{<string,Func\text{<int,Set\text{<int>>>}}}();} \\
  \text{Func\text{<int,int,int>} & f = (x, y) \mapsto x*x + y*y;}
  \end{align*}
  \]

Research impact

- Recent versions of Java and C# show impact of programming language researchers
  - Java: generics, wildcards
  - C#: generics, lambdas, type inference
- Sometimes, perhaps the languages are a little too close to the "bleeding edge" (e.g. it’s an open question whether type checking in Java is decidable!)
- At the other extreme, some languages lack any such underpinnings. A ruby puzzler: what is the value of local variable (initially 0) \( x \) after executing this code?
  \[
  \begin{align*}
  x & = 0 \\
  [1,2,3].each{|x| \text{print x }
  }
  \end{align*}
  \]

Is C# my favourite programming language?

- No. I'm still rather attached to ML.
  - C# has borrowed many features from other languages but the features are sometimes watered down or interact badly with existing features
    - E.g. type inference in C# is limited to local variables and lambda parameters only, and is purely local – no unification.
    - no proper support for separating interface from implementation (cf ML signatures) or parameterization in-the-large (cf ML functors)
  - no algebraic datatypes or pattern matching
Want to know more?

- Original paper on “GJ” (Generic Java):  
  Making the future safe for the past: Adding Genericity to the Java Programming Language. Bracha, Odersky, Stoutamire, Wadler, OOPSLA’98.

- Our paper on .NET generics:  

- Flavour of the moment in functional programming: “Generalized Algebraic Data Types”. (Mostly) possible in C#!  
  Generalized Algebraic Data Types and Object-Oriented Programming. Kennedy & Russo, OOPSLA’05.