Language Interoperability on “dot” NET

Kris Gybels

Vrije Universiteit Brussel
Introduction
• Microsoft’s development platform for “rapid development of Internet and Intranet applications”
• Many similarities with Java (evolved from Microsoft’s Java projects)
• Big difference with Java platform:
  – Java: “write in Java, run everywhere”
  – .NET: “run on Windows, write in any language”
Common Language Runtime

“run on Windows, write in any language”
How Common?

(Announced) Supported Languages:

- APL
- Microsoft JScript
- C#  Microsoft Visual Basic
- C++  Mondrian
- COBOL  Oberon
- Component Pascal  Oz
- Curriculum  Pascal
- Eiffel  Perl
- Forth  Python
- Fortran  RPG
- Haskell  Scheme
- Java  Smalltalk
- Mercury  Standard ML

!! Multi-paradigm Mixture of Languages !!

Note: language developers participated in project to give feedback to .NET team
“.NET does not support 100 different languages, it supports C# with 100 different syntaxes”

Early EIFFEL# != EIFFEL (no support for multiple inheritance)

Current EIFFEL# == EIFFEL (though with few small exceptions)
Language Interoperability

Message interchange between objects

= +

Extension of class in different language
Common Intermediate Language

C# Program

Smalltalk Program

Mercury Program

CIL Type Definitions

CIL Type Definitions

CIL Type Definitions

Common Language Runtime

JIT compilation

CPU
**Common Intermediate Language**

CIL

=! union of supported languages
=! lowest-common denominator of supported languages
= ‘object-oriented assembly’

```csharp
assembly DemystifyingILChapter1 {}
.class public auto ansi HelloWorld extends [mscorlib]System.Object
{
    .method public hidebysig static void HelloWorld() cil managed
    {
        .entrypoint
        ldstr "Hello World."
        call void [mscorlib]System.Console::WriteLine(class System.String)
        ret
    }

    .method public hidebysig specialname rtspecialname
    instance void .ctor() cil managed
    {
        ldarg.0
        call instance void [mscorlib]System.Object::ctor()
        ret
    }
}
```
Common Language Specification

**CLS compliant code**

1. Thou shalt follow Annex 7 of TR 15 of Unicode Standard for names
2. Thou shall not overload fields
3. Thou shall not use unsigned integers

... List of guidelines to follow for *public* classes, methods, fields ... to ensure they are accessible from all languages on the CLR

**Message interchange between objects**

**Extension of class in different language**

**CLS compliant tools / compilers / ...**

*Consumer* level compliance allows only interchange
*Extender* level compliance allows also extension
LI really occurs at level of CIL

mappings of ‘exotic’ language features needs to be usable from other languages
Languages on the CLR

Bit too much guess work involved
class Set {
    // (items variable, constructor, ...)

    public delegate void MapFunc(Object element);

    public void Map(MapFunc func) {
        for (Int32 item = 0; item < items.Length; item++)
            func(items[item]);
    }
}

class AddressBook {
    Set addresses;

    public void PrintAddress(Object address) {
        // cast and print ...
    }

    public void PrintAllAddresses() {
        addresses.Map(new Set.MapFunc(PrintAddress));
    }
}
public class MapFunc : System.MulticastDelegate {
    public MapFunc(Object target, Int32 methodPtr) runtime managed;

    public void virtual Invoke(Object elements) runtime managed;
}

class Set {
    public void Map(MapFunc func) {
        for (Int32 item = 0; item < items.Length; item++)
            func.Invoke(items[item]);
    }
}
- In languages without first-class functions the mapping is revealed
- Functions in other languages *should* be mapped the same way

Convention: perpetual non-believer X-Files agent Dana Scully is used as a label for boxes offering ‘the skeptic’s viewpoint’ on LI in .NET

There is no “Common Mapping Specification”

Smalltalk# uses a different mapping!
Eiffel

Exotic:
- multiple inheritance
- pre/post conditions
- covariance
- generics

Mapping must still allow for
A varA = new Z()

Interfaces mimicking the multiple inheritance structure

Flattened class hierarchy in separate namespace ‘Impl’
+ namespace ‘Create’ for creating instances

Mapping results really in
A varA = new Create.Z()

Can't simply replace e.g. Smalltalk collection hierarchy with an improved multiple-inheritance-exploiting EIFFEL one!
eat(Food x) {
    if (instanceof(x, Grass))
    ...
}

Covariance:

Generics:
S# Smalltalk-based scripting language, still not released

*S*Smalltalk* `toy` implementation by J. Brant and D. Roberts

- CLS consumer-level compiler: not possible to extend .NET classes
- Blocks are mapped to special classes, but != delegates!

```plaintext
find: anObject
  ^ someCollection detect: [:each | each = anObject]

class Block1 {
  Object anObject;
  Block1(Object anObject_);
  value(Object each);
}

find(Object anObject) {
  someCollection.detect(new Block1(anObject));
}
```

Exotic:
- keyword syntax
- dynamic typing
- blocks (& non-local returns)
- meta classes
- more advanced reflection

asDelegate: (aDelegateType) must be explicitely sent to block to pass as delegate to other .NET code
Non-local returns from blocks are simulated with exceptions

These exceptions can be accidentally caught in other languages with try-catch blocks that simply catch every exception.
• Consumer-level due to own object hierarchy:

← Has dummy method for each message in program, dummy method calls ‘doesNotUnderstand:’
• Interaction with non-Smalltalk:

new X.Y.Class() ⇔ X.Y.Class new

object.x(.., .., .., ..) ⇔ object x: .. with: .. with: .. with: ..

aDotNetHashTable := System.Collections.Hashtable new.
aDotNetHashTable add: .. with: ..

↑ invokes add(.., ..)

Smalltalk objects can only be passed as argument to Object type arguments of other CLI methods due to own object hierarchy

A programmer-defined conversion mechanism is provided though
Exotic:
- patterns instead of classes
- not ‘super’, but ‘inner’

Combines semantics of overriding (super) and extending (inner)
A
deposit: (# amount: @integer
    enter amount
    do balance + amount -> balance
    exit balance
#)

A
int deposit(int amount) {
    deposit D = new deposit(this);
    D.enter(amount);
    D.do();
    return D.exit();
}

class deposit {
    int amount;
    void enter(int a) { amount = a; }
    void do() { balance = balance + amount; }
    int exit() { return balance; }
}
A
int deposit(int amount) {
  deposit D = new deposit(this);
  ...
  D.do();
  ...
}
class deposit {
  ...
  void do() { ...; do_1(); ... }
  ...
}

B
int deposit(int amount) {
  deposit D = new deposit(this);
  ...
  D.do();
  ...
}
class deposit extends A.deposit {
  ...
  void do_1() { ... }
  ...
}
Expected

```
x
  inner

x
  super.x()
```

Map of patterns and inner breaks when interoperating

```
class x {
  do() { ... }
}
```

```
class x {
  do_1() { ... }
}
```

```
class x {
  do() { ... }
}
```

```
x
  x.do();
```

```
x
  x.do();
class x {
  do_2() { ... }
}
```

```
x
  super.x();
```

```
x
  super.x();
```

```
x
  super.x();
```

```
x
  super.x();
```
Seems to use a rather primitive “mapping” approach to interoperation.

Exotic:
- functional/logic paradigm

Need to declare a type alias for any CIL class that will be used:

```mercury
:- type xmldoc.
:- pragma foreign_type(il, xmldoc, “class [System.xml]XMLDoc”).
```

“Foreign code” can be embedded in Mercury as a procedure:

```mercury
:- pred load(string::in, xmldoc:di, xmldoc:uo) is det.

:- pragma foreign_proc(”C#”, load(File::in, XMLIn::di, XMLOut::uo),
   [will_not_call_mercury, promise_pure],
   “
   XMLIn.LoadXML(File);
   XMLOut = XMLIn;
   ”
```

Foreign Code -> Mercury: only deterministic predicates, no mention about .NET support??!
Conclusion

• “Message Interchange” (== Linguistic Symbiosis @PROG) works, sort of

• “Class Interchange” problematic
• Too centred on OO & static typing
  “.NET supports C#, and some other languages, badly”
• No “Common Mapping Specification” ?

Follow-up?
• Describe in Patrick Steyaert’s ‘linguist symbiosis’ & reflection framework?
• Pinpoint problems as protocol differences in that framework?