Universidad de Oviedo

Departamento de Informática

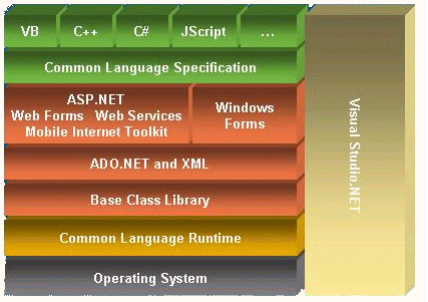
Reflexionando sobre Microsoft .net

Francisco Ortín Soler,
Universidad de Vigo
Ourense, Marzo de 2005

Lenguajes y Sistemas Informáticos

¿Qué es .net?

- El desarrollo de aplicaciones software es una **tarea** en sí **compleja**
  - La apertura de éstas a entornos distribuidos (Internet) la hace **todavía más compleja**
- Microsoft .net es una **plataforma que ofrece la infraestructura** para resolver los problemas comunes del desarrollo de aplicaciones (distribuidas)



.net Framework

- **.net Framework** proporciona soluciones prefabricadas para el desarrollo de aplicaciones distribuidas. Consta de:
  1. Common Language Runtime (**CLR**)
     - Proporciona el **entorno de ejecución** de todas las aplicaciones del sistema
     - Proporciona una capa intermedia entre el sistema operativo y las aplicaciones
     - Su diseño se basa en el concepto de **máquina virtual** o abstracta
  2. Librerías de clases: *Base Class Libraries* (BCL), WinForms, Win32, ADO.NET...
  3. Plataforma de Aplicaciones Web: ASP.NET y Servicios Web

CLR

- La utilización del CLR aporta las siguientes ventajas:
  1. Disponibilidad de las características de cualquier lenguaje: La **elección del lenguaje** estará **condicionada por el dominio del problema**, no por otras características –librerías, eficiencia, *frameworks*... ¿Puedo utilizar el API de Java desde otros lenguajes?
  2. **Heterogeneidad**: Internet es una arquitectura heterogénea, las aplicaciones deberían poder ejecutarse de forma independiente a la plataforma
  3. **Interoperabilidad** de aplicaciones orientadas a objetos, **independientemente de los lenguajes** de programación seleccionados (sistema de tipos común)
  4. **Gestión automática de memoria**: la gestión de memoria es compleja en aplicaciones de servidor

CLR (II)

5. El mantenimiento de aplicaciones da lugar a versiones de componentes OO utilizados por diversas aplicaciones. La **coexistencia de distintas versiones** de componentes demandada por distintos clientes debe ser gestionada sin incoherencias (DLL *hell*)
6. Las aplicaciones distribuidas por redes son cada vez más un canal de adquisición y ejecución de software. La **limitación de acceso a recursos** por aplicaciones posee mayor veracidad que un sistema "todo o nada" de certificados
7. **Organización de las funciones del sistema** operativo mediante jerarquías (espacios de nombres)
8. Debe mantenerse "compatibilidad hacia atrás" para crear e interactuar con código existente (***legacy code***)

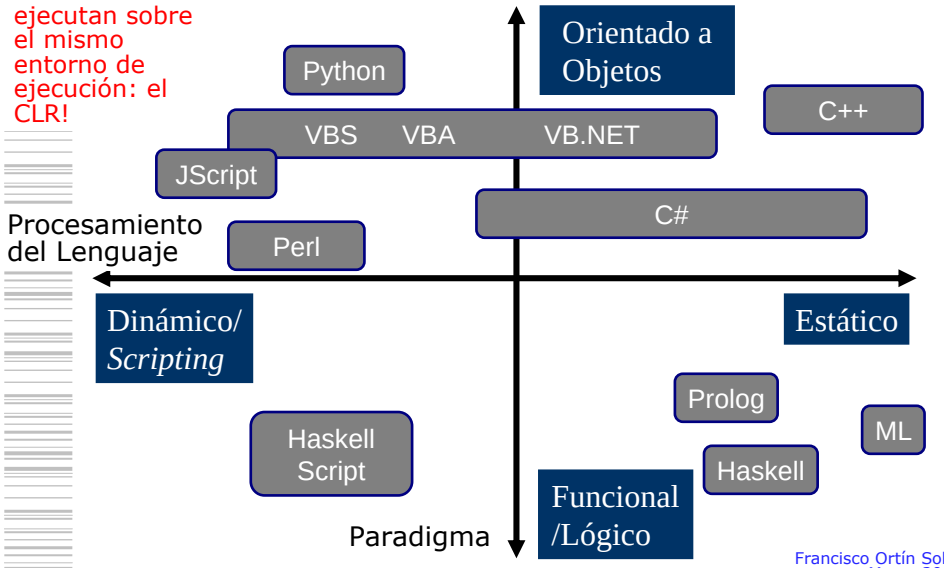
Expresividad de los Lenguajes

- Siempre puede aparecer la **controversia** acerca de qué lenguaje de programación utilizar
- Lo ideal es que la elección de un lenguaje estuviera **condicionada al dominio del problema** (parcial o total) a resolver
- Las facetas propias de:
  - Librerías
  - Gestión de memoria
  - Eficiencia
  - Tamaño de los programas a utilizar
  - Componentes
  - Portabilidad
  - Distribución
  - Persistencia

  estarán **ligadas al entorno de ejecución** (al CLR), no al lenguaje

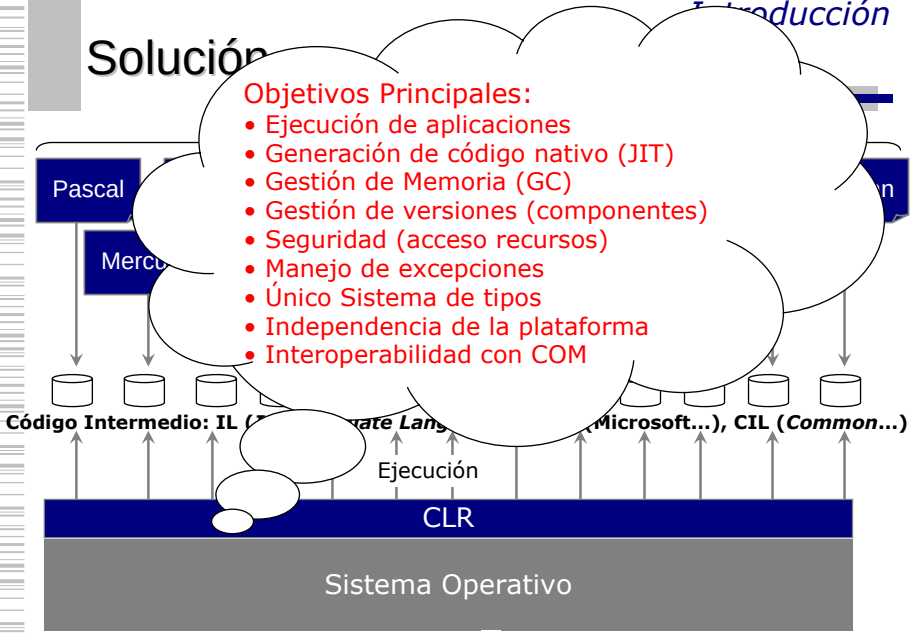
Entorno Computacional Único

¡Todos se ejecutan sobre el mismo entorno de ejecución: el CLR!



Solución

Objetivos Principales:
- Ejecución de aplicaciones
- Generación de código nativo (JIT)
- Gestión de Memoria (GC)
- Gestión de versiones (componentes)
- Seguridad (acceso recursos)
- Manejo de excepciones
- Único Sistema de tipos
- Independencia de la plataforma
- Interoperabilidad con COM

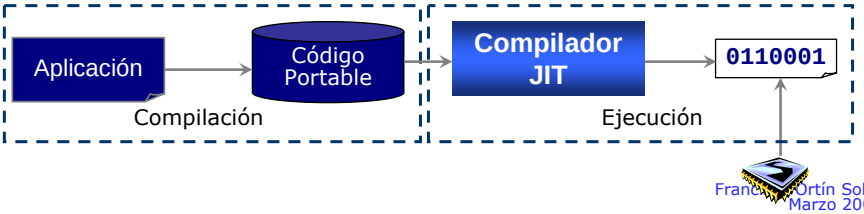


Demo

- HolaVB.vb
- HolaCS.cs
- HolaIL.il
- ILDASM (*Intemediate Language Disassembler)*
- Comparación de código generado

Eficiencia

- La utilización de un único motor computacional (CLR) proporciona multitud de ventajas, pero un claro inconveniente: la **eficiencia**
- La **interpretación** es más **cara** que la ejecución nativa, puesto que existe un nivel computacional más
- ¿Es el CLR un intérprete? **NO**
- En lugar de interpretar el código, el CLR (**Compilador JIT**):
  - **Compila** dinámicamente el código a su representación nativa en la plataforma de ejecución
  - **Optimiza** el código generado (e.g., ***inlining***)



Contenido

- Introducción
- **SSCLI (Rotor)**
- Reflectividad
- Incluyendo Reflectividad Estructural
- Desarrollo del Proyecto
- Documentación

Implementaciones Libres

- Existen diversas **implementaciones libres** del CLI (*Common Language Infrastructure*) ECMA-335 y de C# (ECMA-334)
  - Formato del fichero exe (metadatos y código IL)
  - El sistema de tipos común
  - La especificación extensible (atributos) de metadatos
  - El lenguaje intermedio
  - Acceso a la plataforma host (PAL)
  - La librería básica de clases (BCL)
  - Sintaxis y semántica de C#
- Shared Source CLI (SSCLI, aka Rotor)
  - Implementación de Microsoft libre para investigación y docencia (Windows, FreeBSD y Mac OS x)
- Mono (www.mono-project.com)
  - Implementación libre (liderada por Novell) de los dos estándares ECMA para Linux, Windows, BSD y Solaris
  - Además ofrece otros componentes del .net Framework (ADO.net, Windows Forms, ASP.NET)
- DotGNU Portable.NET (www.dotgnu.org)
  - Implementación de ECMA 335 y 334 para Linux, Windows, [Net,Free]BSD, Solaris y Mac OS X

Componentes de Rotor

Request for Proposals

- Microsoft ha lanzado dos "**peticiones de ideas**" de utilización de Rotor en el campo de la investigación y docencia
  - En abril de 2002 pone a disposición su código fuente y hace una llamada de peticiones (**RFP 1**)
    - Premia a 43 proyectos (con fondos económicos) para desarrollar un proyecto de 18 meses
    - 2 de estos proyectos son españoles
      - Enric Pastor, U. Politécnica de Cataluña, "*.NET (Rotor) in embedded processors*"
      - Marian Díaz, U. Oviedo, "*Adding low-level pure capability-based protection to the CLR for supporting flexible and secure embedded and mobile applications*"
  - En noviembre de 2003 se lanza el **RFP2**
    - En febrero de 2004 premia 38 proyectos,
    - 2 españoles
      - Darío Álvarez, U. Oviedo, "*On the cost of securing applications: Performance and feasibility of capability-based security in the Rotor platform*"
      - Francisco Ortín, U. Oviedo, "*Extending Rotor with Structural Reflection to support Reflective Languages*"

Proyecto RFP2

- El proyecto *Extending Rotor with Structural Reflection to support Reflective Languages* tiene como principal **objetivo**
  - Añadir al motor computacional de .net (Rotor) **reflectividad estructural** para
    - Aumentar la adaptabilidad de la plataforma (nuevas funcionalidades de la BCL, ADO.NET, AOP dinámica…)
    - Dar un soporte nativo (no interpretado) a los, cada vez más utilizados, lenguajes dinámicos (Python, Ruby, Dylan)
    - Aumentar la eficiencia de las implementaciones existentes, gracias a la utilización del JIT *compiler*
    - Permitir la interacción directa con cualquier lenguaje de programación "X.net"

Reflectividad (Reflexión)

- Capacidad de un sistema computacional para **modificar** y **actuar acerca de sí mismo**
  - El **dominio** computacional de un sistema es ampliado con su **propia representación**
- La reflectividad puede producirse en tiempo de compilación (AOP) o en tiempo de ejecución
- Niveles:
  - **Introspección**. Conocimiento dinámico de la estructura de una aplicación (Java, C# o RTTI del C++)

    Ejemplo: Conocer en tiempo de ejecución todos los métodos de una clase e invocar a aquellos que comiencen con **test** (JUnit)
  - **Reflectividad estructural**. Modificación dinámica de la estructura de una aplicación (SmallTalk, Python)

    Ejemplo: Añadir a un objeto un método descrito por el programador en tiempo de ejecución (Smalltalk)
  - **Reflectividad computacional**. Modificación dinámica de la semántica del sistema (MOPs, MetaXa, nitrO)

Introspección en .Net

- La plataforma .net incluye en su BCL dos espacios de nombres dedicados a introspección
  - **System.Reflection**: Introspección dinámica de los ensamblados, clases, objetos, métodos, atributos y propiedades

```
class ClassSuma  {
  public static int sumar(int a,int b) { return a+b; }
  public static void testSuma() {
         int a=4,b=-4;
         Debug.Assert(a+b==sumar(a,b));
 }
  [STAThread] static void Main() { pruebas(typeof(ClassSuma)); }
  public static void pruebas(Type clase) {
         MethodInfo[]methods=clase.GetMethods();
         int n=0;
         for (int i=0;i<methods.Length;i++)
                if (methods[i].Name.StartsWith("test")) {
                       methods[i].Invoke(null,null);
                       n++; }
         Console.Write("{0} pruebas ejecutadas
                correctamente",n);
} }  }
```


Introspección en .Net (II)

- **System.Reflection.Emit**: Generación dinámica de código (IL)
  - Es posible crear programas que generen programas (programación generativa)
    - Con la clase **TypeBuilder** se crean nuevas clases
    - Con la clase **MethodBuilder**, nuevos métodos
    - Con la clase **FieldBuilder**, nuevos atributos
    - Con la clase **ILGenerator** se genera el código IL de un método recién creado
  - Está **limitado a nuevos tipos y métodos**
    - ¡No se puede añadir métodos ni atributos a tipos ya existentes!
    - Tampoco se pueden eliminar

Incluyendo Reflectividad Estructural

- Nuestro proyecto se basa en, haciendo uso del diseño actual de **`System.Reflection`** y **`System.Reflection.Emit`**,
  - Poder
    - añadir
    - eliminar
    - modificar

  atributos y métodos (propiedades) de cualquier
    - clase
    - objeto

  en tiempo de ejecución

Ejemplo de RE en Python

```
class Point:
  "Constructor"
  def __init__(self, x, y):
    self.x=x
    self.y=y

  "Translate Method"
  def translate(self,
          relx, rely):
    self.x=self.x+relx
    self.y=self.y+rely

  "Draw Method"
  def draw(self):
    print "("+str(self.x)+
          ","+str(self.y)+")"

point=Point(1,2)
point.draw()    # (1,2)
```

```
# Modify attributes of
# a single object
point.z=3
print point.z      # 3

# Modify methods of
# a single object
def draw3D(self):
  print "("+str(self.x)+
        ","+str(self.y)+
        ","+str(self.z)+")"

point.draw3D=draw3D
point.draw3D()      # (1,2,3)

# Modify methods of a class
def getX(self):
  return self.x

Point.getX=getX
print point.getX()  # 1
```

Modificación dinámica de estructuras

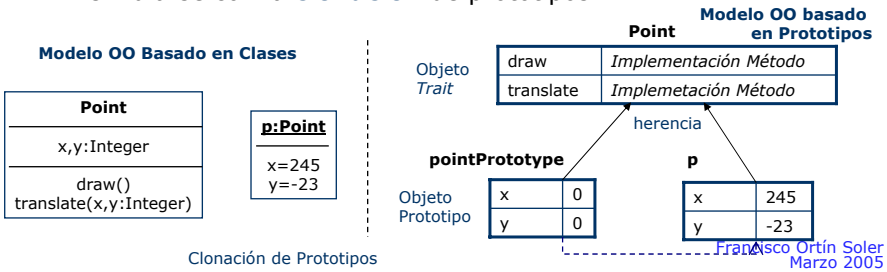
- El añadir reflectividad estructural a un modelo computacional basado en clases da lugar a una serie de escenarios
  1. Modificar la **estructura de una clase** ⇒ Evolución del esquema
     - ✓ En la adición de atributos, se añaden valores por omisión de los mismos
     - ✓ Se utiliza el concepto de excepción para controlar los métodos o atributos obsoletos
  2. Modificar la estructura de **un solo objeto** ⇒ Versiones del esquema (una nueva clase para ese nuevo objeto)
     - ✘ Identidad de las clases (tipos)
     - ✘ Consumo de memoria
     - ✘ Consistencia del modelo…

     … Implica una implementación compleja de un modelo incoherente (abandono del MetaXa)

Francisco Ortín Soler
Marzo 2005

Modelo OO basado en Prototipos

- El modelo computacional OO basado en prototipos (Self o Mootstrap)
  - No utiliza el concepto de clase (**sólo objetos**)
  - El comportamiento compartido por un conjunto de objetos puede agruparse en objetos de comportamiento (***trait objects***)
  - La estructura común de objetos puede definirse mediante objetos **prototipo**
  - La instanciación de objetos (modelo de clases) puede simularse con la **clonación** de prototipos

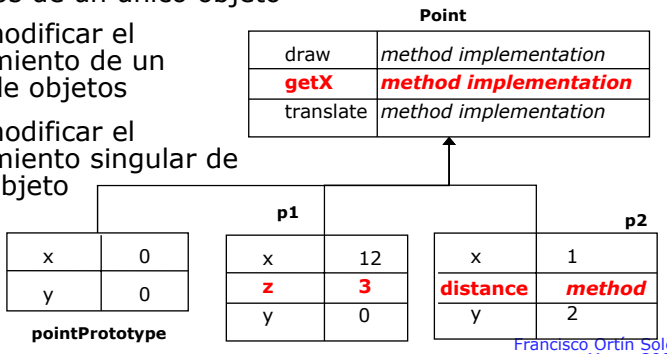


Clonación de Prototipos

Francisco Ortín Soler
Marzo 2005

Versiones de Clases

- El modelo orientado a objetos basado en prototipos soluciona los problemas planteados con las versiones (y evolución) de esquemas
- Es posible modificar la estructura de un único objeto **de un modo consistente**
  - Permite modificar la estructura de atributos de un único objeto
  - Permite modificar el comportamiento de un conjunto de objetos
  - Permite modificar el comportamiento singular de un único objeto



Reflectividad Estructural en Rotor

- Pero, ¡¡¡ Rotor (.net) utiliza un modelo computacional **basado en clases** !!!
- Es necesario que se siga soportando el modelo basado en clases para:
  1. Que las aplicaciones existentes sigan funcionando
  2. Que exista compatibilidad e interoperabilidad entre aplicaciones y lenguajes
- El programador seguirá empleando el concepto de clases, pero **al emplear los servicios de reflectividad estructural**:
  - Las clases serán tratadas como objetos *trait* (comportamiento compartido)
  - Los objetos serán capaces de representar tanto atributos como métodos (*slots*) evitando la evolución del esquema

  Internamente traduciremos el modelo de Rotor a un modelo **basado en prototipos**

Sistemática

- El método que hemos empleado para cumplir los objetivos ha sido:
  1. Desarrollar una "**versión BCL**" para detectar incompatibilidades e incoherencias de implementación
     - Ampliar la BCL con toda la funcionalidad de las primitivas de Reflectividad Estructural
     - Acceder al 100% de los servicios mediante la BCL
     - Evaluar las inconsistencias y rediseñar el modelo
     - Evaluar el rendimiento
  2. Implementación de todas las primitivas dentro de la máquina virtual (*runtime environment*), en el lenguaje C
  3. Añadir a todo el juego de **instrucciones IL** la nueva semántica del modelo
     - Añadir la nueva semántica de un modo ortogonal a las instrucciones IL (**`ldfld`**, **`stfld`**, **`callvirt`**, **`call`**…)
     - Generar el código nativo de dicha semántica mediante el empleo del JITter
     - Evaluar el rendimiento en comparación con implementaciones similares
  4. Evaluar las posibilidades de *mapping* de los **lenguajes dinámicos** existentes al SRSSCLI

Versión BCL

- Hemos ampliado la BCL con el espacio de nombres **`System.Reflection.Structural`**
- El conjunto principal de primitivas que posee son métodos estáticos de la clase de utilidad **`Structural`**:
  - **`addMethod`** y **`removeMethod`**: reciben un objeto o una clase (**`Sytem.Type`**) y la descripción de un método (**`MethodInfo`**).
    - Los métodos pueden copiarse de otra clase o generarse dinámicamente mediante **`System.Reflection.Emit`**
  - **`invoke`**: Ejecuta un método de un objeto o una clase. Implementa el mecanismo de herencia.
  - [**`add`**,**`get`**,**`remove`**]**`Field`**: Modifica la estructura de atributos de un objeto o clase.
- Hemos creado otras primivitas y clases adicionales, de menor importancia

Ejemplo C#

- Lo siguiente es un ejemplo en C# de utilización de la versión BCL

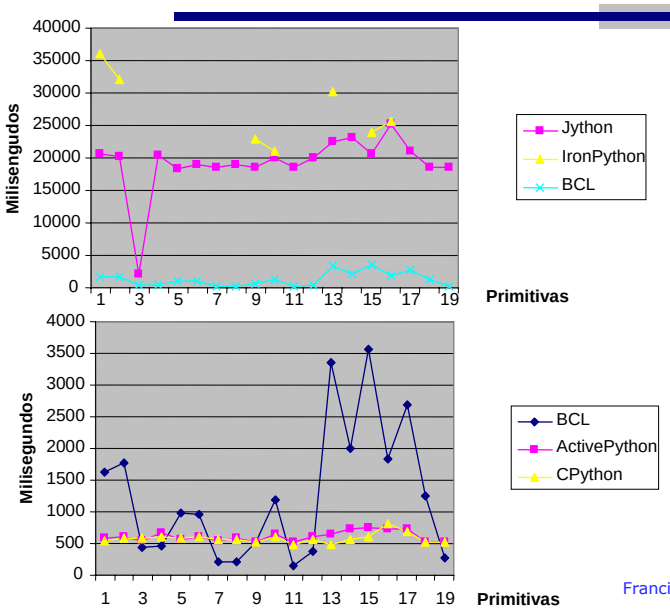
```
RuntimeStructuralFieldInfo rsfi = new RuntimeStructuralFieldInfo(
   "z", typeof(int),3, FieldAttributes.Public);
Structural.addField(point,rsfi);
// * Draw3D es un objeto MethodInfo creado con
//      System.Reflection.Emit
Structural.addMethod(point,draw3D);
Object[] pars={};
Structural.invoke(point,draw3D.ReturnType, "draw3D",pars);
// * getX is otro objeto MethodInfo
Structural.addMethod(typeof(Point),getX);
Console.WriteLine(Structural.invoke( point, getX.ReturnType,
   "getX",pars) );
rsfi = new RuntimeStructuralFieldInfo( "isShowing", typeof(bool),
   false, FieldAttributes.Public);
Structural.addField(typeof(Punto), rsfi);
```


Evaluación del Rendimiento

- Hemos evaluado la implementación de la versión BCL de SRSSCLI comparándola con las primitivas de:
  - **CPython** 2.4. En entorno de desarrollo más extendido de Python, desarrollado en C
  - **ActivePython** 2.4 de ActiveState (C)
  - **Jython** 2.1. Compila a código 100% java
  - **IronPython** 0.6. Genera código IL para el CLR
- Las primitivas comparadas fueron:
  - [**add**, **delete**][**method**, **attribute**]
  - **invoke**, **accessField**

  para distintos tipos y escenarios

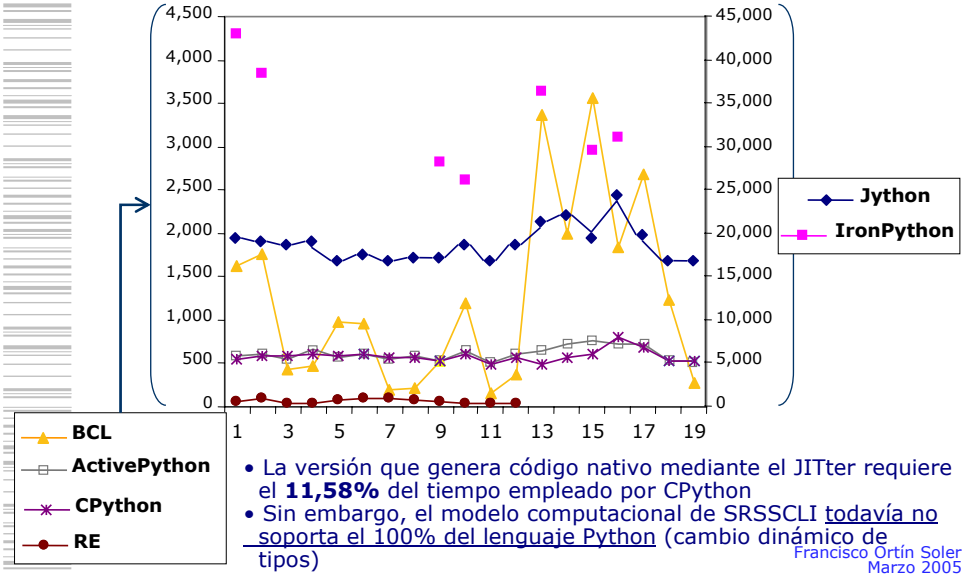
Resultados



Estado Actual

- El estado actual del proyecto se encuentra en la parte central del punto 3
  - Hemos desarrollado el 100% de la versión BCL dentro de la máquina virtual (*runtime environment*) –fase 2.
  - Hemos implementado el control de los atributos de clase y objeto de un modo nativo, mediante el JITter (modificando las instrucciones **`ldfld`** y **`stfld`**)
  - Tenemos parte de la implementación de los métodos (**`callvirt`**); faltan los métodos de clase (**`call`**)
  - Hemos evaluado el rendimiento de lo existente, en comparación con las implementaciones existentes…

Evaluación generación JIT



- La versión que genera código nativo mediante el JITter requiere el **11,58%** del tiempo empleado por CPython
- Sin embargo, el modelo computacional de SRSSCLI todavía no soporta el 100% del lenguaje Python (cambio dinámico de tipos)

Referencias

- Existen dos libros editados sobre el Rotor
  - **Shared Source CLI Essentials**
    David Stutz, Ted Neward, Geoff Shilling
    O'Reilly, 2003.
  - **Distributed Virtual Machines: Inside the Rotor CLI**
    Gary Nutt
    Addison-Wesley, 2004.
- Sobre el código IL
  - **Inside Microsoft .net IL Assembler**
    S. Lidin
    Microsoft Press, 2002
- Sobre el estándar ECMA-335 (CLI)
  - **The Common Language Infrastructure Annotated Standard**
    James S. Miller, Susann Ragsdale, Jim Miller
    Addison-Wesley Professional, 2003

URLs

- Descarga del Rotor
  http://msdn.microsoft.com/net/sscli
- *Request for Proposals*
  - http://research.microsoft.com/Collaboration/University/Europe/RFP/
- .net Framework
  http://msdn.microsoft.com/netframework/
- Especificación de los estándares ECMA
  - http://www.ecma-international.org/publications/standards/Ecma-334.htm
  - http://www.ecma-international.org/publications/standards/Ecma-335.htm
- Lista de proyectos de desarrollo con Rotor
  http://research.microsoft.com/programs/europe/rotor/
- Nuestra web de reflectividad y del SRSSCLI
  http://www.di.uniovi.es/reflection/lab

Francisco Ortín Soler
Marzo 2005

Universidad de Oviedo  Departamento de Informática

Reflexionando sobre Microsoft .net

Francisco Ortín Soler,
Universidad de Vigo
Ourense, Marzo de 2005

Lenguajes y Sistemas Informáticos