DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

UNIVERSIDAD POLITÉCNICA DE VALENCIA

P.O. Box: 22012 E-46071 Valencia (SPAIN)

Informe Técnico / Technical Report

Ref. No: DSIC II/06/05 **Pages:** 12
Title: Integración de componentes Haskell en .NET
Author (s): Beatriz Alarcón y Salvador Lucas
Date: 18 de Julio de 2005
Key Words: COM, Interoperabilidad, .NET, Programación Funcional

VºBº **Author (s):**
Leader of Reasearch Group

Integración de componentes Haskell en .NET

Beatriz Alarcón y Salvador Lucas
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
{balarcon, slucas}@dsic.upv.es

Resumen

.NET es un proyecto emergente de Microsoft para promover un nuevo entorno de desarrollo de software con énfasis en transparencia de redes, con independencia de la plataforma hardware o software (lenguaje de programación, gestor de bases de datos, etc.) y que permita un rápido desarrollo de aplicaciones. El objetivo de este trabajo es investigar la integración de componentes software desarrollados en un lenguaje funcional como Haskell junto con componentes desarrollados en C# u otro lenguaje .NET. Los lenguajes funcionales son muy indicados para desarrollar herramientas de análisis y transformación de programas mientras que los entornos de desarrollo para .NET dan soporte a características avanzadas como asistentes para interfaces gráficas, el proceso de documentos XML y HTML, servicios Web, etc. Para conseguir nuestro objetivo hacemos uso de las facilidades de ofrecidas en .NET para importar componentes COM, por un lado, y la tecnología recientemente desarrollada para generar componentes COM a partir de módulos Haskell.

Palabras clave: COM, Interoperabilidad, .NET, Programación Funcional.

1. Introducción

Durante los más de cincuenta años de historia de la informática ésta ha experimentado un desarrollo indudable. Sin embargo, algunos de los problemas más importantes, como el desarrollo de software fiable sobre bases ingenieriles sólidas, siguen constituyendo un reto que, como ha apuntado recientemente un cientifico tan prominente como T. Hoare, requerirá el esfuerzo combinado de muchos grupos de investigación y equipos de producción de software [Hoa03]. El desarrollo de herramientas de análisis, verificación y depuración automática de sistemas software constituirá una parte importante de este esfuerzo, en el cual la posibilidad de utilizar los lenguajes y técnicas de programación más apropiados para cada propósito será fundamental. Puesto que, tanto desde el punto de vista conceptual como desde el punto de vista práctico, esta metodología de trabajo distribuida es inevitable, se hace necesario avanzar de forma simultánea en las técnicas de interoperabilidad de aplicaciones y herramientas de desarrollo.

Los esfuerzos internacionales por desarrollar un marco global para la utilización de recursos informáticos tienen en Java[1] y en .NET[2] sus más conocidos exponentes. La plataforma .NET permite integrar tecnologías ya existentes, productos modificados para su uso dentro de la plataforma, y elementos nuevos, lo que permite conectar

[1] http://java.sun.com
[2] http://www.microsoft.com/net

información, personas, sistemas y dispositivos a través de software. También el proyecto XML auspiciado por el WWW consortium[3] está relacionado con este esfuerzo a través del uso en .NET de XML para documentar programas, el soporte de servicios Web basados en XML, etc. Como atestiguan los numerosos congresos y talleres dedicados recientemente a explorar este tipo de problemas, hay una necesidad urgente por parte de la comunidad científica que investiga en el desarrollo y aplicación de métodos formales en la Ingeniería del Software de encontrar una manera simple y sistemática de combinar herramientas de análisis y desarrollo de software. Es interesante estudiar hasta qué punto las plataformas mencionadas podrían proporcionar un marco válido o interesante para el desarrollo y uso distribuido pero coordinado de herramientas de análisis, optimización, compilación, etc.

El trabajo descrito en este informe forma parte de las tareas programadas dentro del proyecto SELF[4] cuyo principal objetivo es contribuir a ese reto común. Las comunidades científicas que desarrollan lenguajes y tecnología software declarativa están realizando un esfuerzo serio para integrarse en este tipo de iniciativas. Como parte de dicho esfuerzo, se han desarrollado herramientas de análisis de programas como MU-TERM [Luc04] y MTT (Maude Termination Tool) [DLMMU04] que utilizan herramientas externas auxiliares como CiME[5] o el propio compilador del lenguaje Maude[6] empleando distintos mecanismos de interoperabilidad, desde simples llamadas directas al sistema operativo hasta servicios Web. La comunidad internacional ha recibido con interés este tipo de iniciativas, pero queda mucho por hacer en los distintos frentes involucrados: definición de formatos de intercambio de información, inclusión de puntos de entrada adecuados en las herramientas desarrolladas, etc.

Nadie duda de la gran potencia de los lenguajes funcionales, ni de la facilidad con la que se pueden expresar funciones que en los lenguajes imperativos serían más complejas. Las facilidades para el ajuste de patrones y las características de orden superior hacen a los lenguajes funcionales como Haskell[7] muy indicados para el análisis y transformación de programas. Sin embargo, el talón de Aquiles de Haskell puede que sea la inexistencia de una herramienta visual para la realización de interfaces gráficas de manera rápida, como poseen, por el contrario, muchos otros lenguajes de programación. Así, un programador Haskell puede perder demasiado tiempo en darle forma a su aplicación mientras que con un entorno de desarrollo integrado (IDE) como Visual Studio .NET, esto resulta sencillo. El soporte para la definición de servicios Web que ofrece la plataforma .NET es otra ventaja de la que se podrían beneficiar las aplicaciones Haskell.

En el mundo de los lenguajes funcionales, existen adaptaciones más o menos completas a .NET de los lenguajes:

ML	http://www.cl.cam.ac.uk/Research/TSG/SMLNET
Mondrian	http://www.mondrian-script.org

[3] http://www.w3.org
[4] http://self.lcc.uma.es
[5] http://cime.lri.fr
[6] http://maude.cs.uiuc.edu
[7] http://haskell.org

Por lo que respecta a Haskell, las iniciativas existentes son todavía poco maduras[8], pero revelan el interés de la comunidad por converger a la plataforma .NET. En 1999 un grupo de investigadores exploraron la posibilidad de encapsular los programas Haskell como objetos COM (*Microsoft's Component Object Model*). ¿Por qué no podiamos dar un paso más y aprovechar la interoperabilidad de COM con .NET para lograr nuestro objetivo? Microsoft ha dejado abierta la posibilidad de usar los componentes COM ya existentes en .NET; así, un programador de Windows no deberá reescribir todas sus aplicaciones ante la llegada de .NET. En nuestro caso, podemos beneficiarnos de esto para empaquetar nuestros programas Haskell como componentes software e integrarlos en aplicaciones escritas en otros lenguajes, por ejemplo en C#, el lenguaje por excelencia de .NET.

Nuestro punto de apoyo es HaskellDirect[9] (HDirect [Fin99,FLMP99]) un marco de trabajo de *interfaz para funciones externas* (FFI, *Foreign Function Interface*) para Haskell basada en el estándar IDL (*Interface Definition Language*) permite especificar una interfaz de programación de manera independiente del lenguaje. A partir de él, profundizaremos en el objetivo que nos ocupa: la interoperabilidad de Haskell con .NET a través de COM.

Son muchas las posibilidades que ofrece HDirect:

- Creación de enlaces Haskell con librerías externas.
- Creación de enlaces externos con librerías Haskell.[10]
- Creación de interfaces cliente Haskell para objetos COM.
- Creación de componentes COM en Haskell.

Para el fin que nos ocupa, el punto que más nos importa es el de construir un objecto COM en Haskell capaz de interoperar con aplicaciones Windows y con ello conseguir nuestro principal objetivo: interoperar con .NET. Para más información sobre HDirect puede consultar el extenso manual de éste [Fin99] o [Hdi] donde se explican los detalles de su implementación que no consideramos prioritarios para este documento.

En la sección 2 introducimos nuestro caso de estudio y mostramos cómo construir un componente COM a partir de un módulo Haskell. En la sección 3 abordamos la problemática de su integración en .NET. La sección 4 presenta nuestras conclusiones y líneas de investigación futuras.

2. Interoperabilidad mediante COM en Haskell

La tecnología de Microsoft COM es usada por los desarrolladores para crear componentes reutilizables, enlazar componentes para construir aplicaciones, tener las ventajas de los servicios Windows y permitir a los componentes software comunicarse.

[8] ver http://www.haskell.org/hugs y http://galois.com/~sof/hugs98.net .
[9] http://www.haskell.org/hdirect/

Para abordar el problema pensamos en un ejemplo muy sencillo que nos abstrayera de detalles de implementación. Creamos una simple aplicación Windows en Visual Studio.Net 2003 con un botón y una caja de texto. Nuestro deseo era que al pulsar el botón apareciera en mayúsculas la cadena en la caja de texto. La interfaz gráfica la hemos implementado en C# mientras que la trasformación la realiza un módulo Haskell mediante la llamada a la siguiente función:

toMay st = return (map toUpper st)

2.1. Módulos Haskell y componentes COM

Un programa Haskell que implementa un componente COM consiste en tres partes diferenciadas:

- El código de la aplicación Haskell escrito por el programador.
- Un conjunto de modulos Haskell generados automáticamente a partir de la IDL y la herramienta HDirect que trataran la conexión entre Haskell y COM.
- Un módulo de librería Haskell, *Com*, el cual exporta todas las funciones necesarias para soportar objectos COM en Haskell y un modulo de librería en C que proporciona soporte en tiempo de ejecución (*RTS, run-time support*).

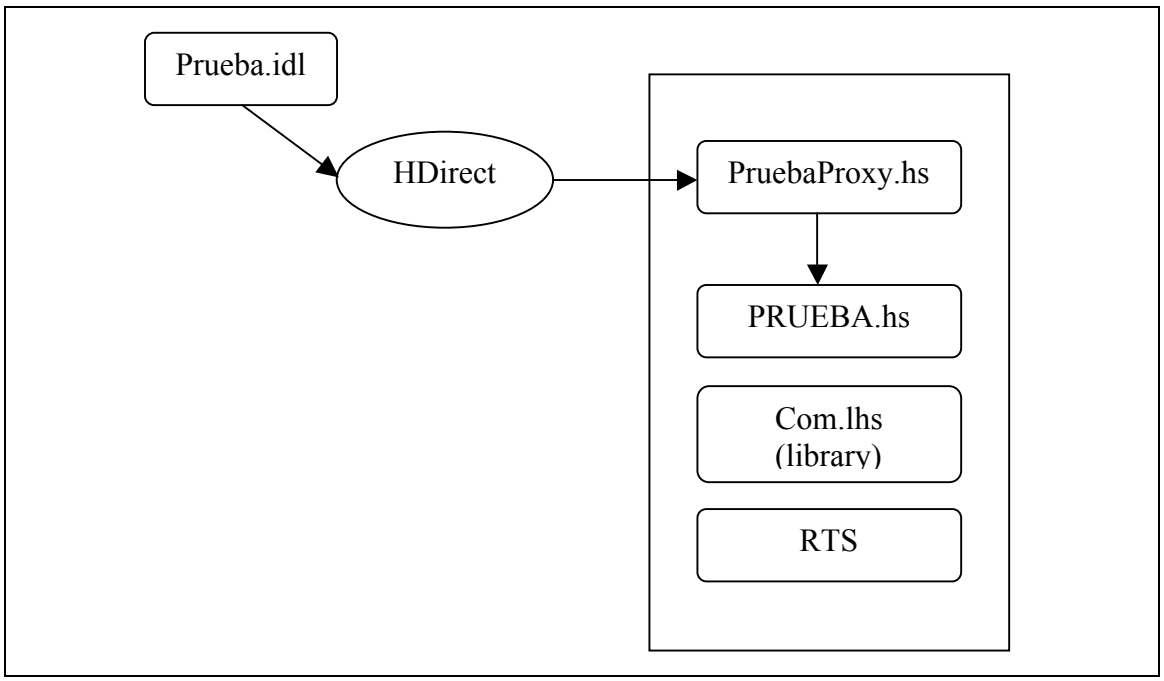

Figura 1. Un componente COM en Haskell

Empezamos bajándonos la última versión de HDirect 0.21 en código fuente. Existía una versión más antigua compilada para Windows (0.16) pero daba problemas con versiones actuales de GHC[10] (*Glasgow Haskell Compiler*) (nosotros hemos utilizado la version de GHC 6.2.2), de hecho, la última versión dio también bastantes problemas a la hora de su instalación. Puesto que la compilación se llevaba a cabo a través de un Makefile, instalamos el programa *cygwin* un emulador de Linux sobre Windows desde es cual hemos trabajado.

[10] http://www.haskell.org/ghc/

2.2. La IDL del componente

El punto de partida es la especificación de las interfaces del componente en IDL. Hemos partido del ejemplo de [FLMP99] y de las indicaciones del manual de HDirect [Fin99] puesto que la información encontrada sobre IDL no ha sido muy extensa ni precisa [Hlu].

El siguiente es el código IDL que implementamos para iniciar el estudio:

```
[ uuid(F0D695CD-51C6-4cd5-81FF-3ECE74FF8F65)
, helpstring("Haskell COM component...")
, version(1.0)
]
library Prueba {

  importlib("stdole32.tlb");

[ object
  , uuid(5F124E73-7553-4e2c-82A9-810129A4CA56)]

interface Iprueba : IUnknown {

   HRESULT toMay([in,string]BSTR in, [out,retval] BSTR *out);
};


  [ object,
   uuid(9D021AE2-21BC-4eff-BE24-41B0BBB2C412)
  ]
  coclass PRUEBA {
    [default]interface Iprueba;

  };
      };
```

A continuación describimos el código IDL anterior el cual servirá al mismo tiempo para entender qué es COM [Ste04,COM].

Mediante del código anterior, estamos formando el esqueleto del objeto que queremos encapsular. Para ello contamos con una librería tipo (*Prueba*), una interfaz (*IPrueba*) y una clase (*PRUEBA*).

Una librería tipo es un archivo binario que contiene la misma clase de información que se encontraría en un archivo de cabecera de C o C++. Contiene los nombres de las clases y de las interfaces que se implementan dentro de un servidor y el número y tipo de parámetros para cada método de sus interfaces. Contiene también los GUID (Globally Unique IDentifiers), parte clave del modelo de programación COM, para cada clase e interfaz. Una estructura de 128 bits creados de tal forma que ningun GUID se repita. En nuestro caso hemos usado la herramienta *Create GUID* que incorpora Visual Studio.NET para generarlos.

Una interfaz COM es una colección de métodos relacionados que llevan a cabo una funcionalidad. Todas están basadas en la interfaz *IUnknown* y todas están identificadas mediante un identificador único de interfaz (IID).

Una clase COM es la implementación de una o más interfaces COM, mientras que un objecto COM es una instancia de una de las anteriores.Cada objeto tiene un identificador globalmente único de clase (CLSID).

Los CLSID y los IID son subconjuntos de los GUIDs.

Con lo anterior quedan justificados los números que aparecen en nuestro código IDL. Nuestra única interfaz se llama *Iprueba* y hereda de *IUnknown*, ya que todas soportan los métodos *QueryInterface, AddRef* y *Release* que implementa Unknown.No se permite la herencia de múltiples interfaces. El primer atributo, *object*, identifica a la interfaz como una interfaz COM. El atributo *in* indica que ese parámetro se utiliza como entrada al método, *out* indica salida. El atributo *string* se utiliza con parámetros que son punteros a caracteres.La palabra clave *retval* indica que ese parámetro no es solo cualquier salida, sino que tiene que interpretarse como el valor de retorno de la función. Debe hacerse así, porque el valor de retorno literal del método es un HRESULT, el cual se utiliza para devolver la información de errores.

2.3 Encapsulando Haskell como COM

Una vez especificada la IDL, el siguiente paso es generar el proxy y el esqueleto de nuestro componente. Un proxy es un programa intermediario que actúa tanto como servidor como cliente, con la finalidad de realizar peticiones en el nombre de otros clientes. Para generar esos módulos utilizamos la siguiente instrucción:

```
ihc -fcom prueba.idl -s --skeleton
```

La cual genera dos ficheros Haskell: *Prueba.hs* y *PruebaProxy.hs.* El primero contiene el esqueleto de los metodos que implementa nuestro componente, es decir la estructura Haskell para los métodos declarados en la IDL. El otro proporciona un proxy que adapta nuestros métodos detrás de una interfaz COM.

El siguiente paso es rellenar el esqueleto con el código Haskell de nuestros métodos. En nuestro caso podría quedar así:

```
- Automatically generated by HaskellDirect (ihc.exe), snapshot 161204
-- Created: 13:40 Hora estándar romance, Friday 08 July, 2005
-- Command line: -fcom prueba8.idl --skeleton

module PRUEBA where

import Prelude (fromEnum, toEnum)
import qualified Prelude
import Char

data State = State

new :: Prelude.IO State
new = Prelude.return (State)

toMay :: Prelude.String
      -> State
      -> Prelude.IO Prelude.String
toMay in0 obj = Prelude.return(Prelude.map toUpper in0)
```

Lo resaltado es lo único que debe hacer el programador (para un método tan sencillo como el que nos ocupa, claro). O incluso podría ser una simple llamada al método implementado en otro módulo Haskell que importáramos.

El siguiente paso es compilar los nuevos archivos:

```
ghc –c Prueba.hs  PruebaProxy.hs
```

Lo cual nos genera los ficheros .hi y .o y los stubs del proxy. Pueden aparecer problemas en la compilación porque *PruebaProxy.hs* necesite módulos de HDirect que no se encuentren en le directorio actual. Para subsanarlos, en el caso de que salga el error "*Type signature given for an expression",* se debe usar la opción *–ffi* al compilar para tener constancia de que módulos busca y dónde.

Una vez llegados a este punto se debe decidir como encapsular nuestro componente. Las librerias de HDirect proporcionan soluciones para construir servidores de proceso interno (DLLs) o servidores de proceso externo (EXEs). Hemos elegido implementar una DLL, que aunque en principio conlleva algo más de esfuerzo, es la forma más sencilla de funcionamiento del modelo COM, ya que la carga del objeto COM se hace de forma transparente al usuario.

Continuando con la idea de implementar una DLL, el siguiente paso consiste en incluir en el directorio los módulos ComDllMain.lhs y dll_stub.c y compilarlos. Finalmente hay que proporcionar un módulo Main por exigencias de GHC sin más importancia que informativa: nombre del componente, versión…

```
module Main(main, comComponents) where


import PruebaProxy
import ComDll


comComponents :: [ ComponentInfo ]
comComponents =
  [ hasTypeLib $
    withComponentName "Haskell Test COM component" $
    withProgID "Haskell.Test" $
    withVerIndepProgID "Haskell.Test.1" componentInfo
  ]

main = putStrLn "In Main"
```

Una vez compilado el Main pasamos a construir la DLL con GHC:

```
ghc --mk-dll –o comdll.dll –optdll—def –optdllComServer.def
<object files> ComDllMain.o dll_stub.o –fglasgow-exts –syslib com
```

Se debe incluir en el directorio *ComServer.def* y en <object files> incluir todos los módulos compilados.

3. Integración de COM en .NET

Llegados a este punto, debemos preocuparnos de insertar la DLL COM en el Visual Studio.NET, objetivo final de nuestro trabajo.

Debido a las múltiples pruebas realizadas y para evitar repetir los mismos pasos para cada una, creamos un *Makefile* para facilitar la experimetación y realización de pruebas.

Las indicaciones para utilizar el *Makefile* son:

En el directorio elegido incluir la especificación IDL, generar los ficheros .hs y el proxy, rellenarlos y compilarlos e incluir ComServer.def, ComPrim.h, dll_stub.c, Main.hs y ComDllMain.lhs.

En *src* tendremos el resto de módulos de HDirect. *Lib* es una carpeta generada automáticamente al compilar HDirect.

A continuación mostramos el código del *Makefile*:

```
TOP = ..
include $(TOP)/mk/boilerplate.mk
HC=ghc

SRC_IHC_OPTS += -s -fanon-typelib
SRC_HC_OPTS  += -fglasgow-exts -syslib com -fno-warn-missing-methods -
v
ifeq "$(FOR_BINARY_DIST)" "YES"
SRC_CC_OPTS  += -I../../src
else
SRC_CC_OPTS  += -I../../lib
endif

HS_SRCS  = Main.hs PRUEBA.hs ComDllMain.lhs PruebaProxy.hs
C_SRCS   = dll_stub.c

SRC_DIST_FILES  = $(HS_SRCS) $(IDL_SRCS) $(C_SRCS) Makefile

# Object files to include into DLL in addition to the $(HS_OBJS)
OBJS     += PruebaProxy_stub.o ComDllMain_stub.o IncTlb.o

# Just export what an self-registering inproc server is supposed to.
DLL_OPTS += -v -optdll--def -optdllComServer.def

CLEAN_FILES += PruebaProxy.hs PruebaProxy.hi $(HS_PROG)$(exeext)
CLEAN_FILES += comdll.dll PruebaProxy_stub.c PruebaProxy_stub.h
prueba.tlb
CLEAN_FILES += ComDllMain_stub.h ComDllMain_stub.c

.PHONY: stubs
stubs : PruebaProxy.hs

prueba.tlb PruebaProxy.hs : prueba.idl
      $(IHC) $(IHC_OPTS)  -v -c prueba.idl -o PruebaProxy.hs --output-
tlb=prueba.tlb
```

```
IncTlb.o : prueba.tlb


IncTlb.o : IncTlb.rc
      windres -i $< -o $@

comdll.dll :: $(OBJS)
      $(HC) --mk-dll -o $@ $(OBJS) $(DLL_OPTS) $(HC_OPTS)
      @echo "Component built ok, make sure you register it (via
regsvr32, for example)"

all :: comdll.dll

include $(TOP)/mk/target.mk
```

Como se indica cuando acaba la compilación, es necesario registrar el componente generado, la manera más sencilla es a través de *regsvr32.exe.* Si bien es cierto que con este comando no llega a realizarse un registro completo del componente y es necesario ir al registro a modificarlo (*regedit.exe*) [Tro02]

COM usa únicamente una rama del registro:HKEY_CLASSES_ROOT. Bajo ésta se lista la clave CLSID. Por debajo de ella, estan listados todos los CLSIDs de todos los componentes instalados en el sistema. Un CLSID está contenido en el registro como una cadena alfanumérica formateada de la manera:{xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}. Cada clave CLSID tiene como valor por defecto el nombre para el componente.

Para cada clase CLSID, existe la subclave *InprocServer32* cuyo valor por defecto es el fichero de la DLL. Estas dos claves constituyen la información más importante del registro. En nuestro caso el registro debería contener las siguientes nuevas entradas:

```
HKEY_CLASSES_ROOT\comdll.PRUEBA\CLSID = {9D021AE2-21BC-4eff-BE24-41B0BBB2C412}

HKEY_CLASSES_ROOT\CLSID\{9D021AE2-21BC-4eff-BE24-41B0BBB2C412}=comdll.PRUEBA

HKEY_CLASSES_ROOT\CLSID\{9D021AE2-21BC-4eff-BE24-41B0BBB2C412}\InprocServer32=
C:\path_to_dll\comdll.dll

HKEY_CLASSES_ROOT\CLSID\{9D021AE2-21BC-4eff-BE24-41B0BBB2C412}\TypeLib={F0D695CD-
51C6-4cd5-81FF-3ECE74FF8F65}

HKEY_CLASSES_ROOT\TypeLib\{F0D695CD-51C6-4cd5-81FF-3ECE74FF8F65}\1.0\0\Win32=
C:\path_to_tlb\prueba.tlb
```

3.1 Utilizando componentes COM en aplicaciones .NET

Un cliente .NET no puede comunicarse directamente con un componente COM porque las interfaces expuestas por el componente COM pueden no ser leídas desde la aplicación .NET. Por ello, para comunicarse con un componente COM, éste debe ser adaptado de modo que la aplicación cliente .NET pueda entenderlo. Esto se realiza mediante un RCW (*Runtime Callable Wrapper*). Los tipos de datos, los prototipos de los métodos y los mecanismos de control de errores no son iguales en los modelos de objetos administrados y no administrados. Para simplificar la interoperación entre los componentes de .NET Framework y el código no administrado y facilitar la ruta de acceso de migración, el CLR (*Common Language Runtime*) oculta las diferencias que existen entre estos modelos de objetos a los clientes y a los servidores. El .NET SDK proporciona RCW para conseguirlo, así permite a una aplicación .NET ver el componente no administrado como si fuera administrado [FPBBG03]. En .NET hay varios modos de realizar esto:

- La utilidad importador de la biblioteca de tipos, o Tlbimp.exe, proporcionada junto a .NET Framework.
- Hacer referencia directamente al COM desde la aplicación de C# VS.NET

Las bibliotecas de tipos pueden encontrarse como archivos individuales; entonces suelen tener extensión TLB.

Las bibliotecas de tipo también pueden encontrarse incrustadas en un servidor COM como un recurso binario. Los servidores COM en curso empaquetados como DLL y los servidores COM detenidos, empaquetados como EXE, pueden incluir la biblioteca de tipos como un recurso del propio servidor COM.

En nuestro estudio hemos llevado a cabo las dos posibilidades anteriormente mencionadas aunque hemos obtenido éxito con la primera. Hablaremos pues de la opción de usar la utilidad *Tlbimp*.

Tlbimp es una aplicación individual de consola que crea el ensamblado de interoperabilidad .NET basado en la DLL COM que especifique. Está situada en el directorio Framework SDK en Archivos de programa.

El comando tlbimp admite el nombre de un archivo de biblioteca como dato introducido: *tlbimp prueba.tlb*

También puede admitir el nombre de un servidor COM que contiene la biblioteca de tipos incrustada: *tlbimp comdll.dll*

Al usar la opción */out* , se puede especificar un nombre alternativo para el ensamblado .NET creado: *tlbimp comdll.dll /out:tlbimpcom2.dll*

El ensamblado producido por la herramienta Tlbimp.exe es un ensamblado .NET estándar que puede ser visto con la herramienta Ildasm.exe.

Despues de registrar nuestra DLL como se ha explicado en el apartado anterior, usamos Tlbimp y nos vamos a VS.NET. Allí creamos una sencilla aplicación Windows con un botón y dos cajas de texto,una en la que insertaremos la cadena que queremos transformar y la otra donde visualizaremos el resultado. A continuación, hacemos click con el botón derecho sobre el archivo *References* en el Explorador de soluciones, seleccionamos *Agregar referencia* y buscamos el ensamblado que acabamos de generar. Una vez que ya está agregado, puede usarse exactamente igual que cualquier otro

ensamblado .NET: creamos una instancia de la clase y ya podemos acceder a los métodos.

```
private void button1_Click(object sender, System.EventArgs e)
{
    //Creamos una instancia de la clase
    tlbimpcom2.PruebaClass p=new PruebaClass();

    //Llamamos a nuestro método toMay
    textBox2.Text=p.toMay(textBox1.Text);
}
```

El código anterior ha sido extraído de nuestra aplicación. Se puede observar como somos capaces de crear una instancia de nustra clase y a partir de ahí llamar a sus métodos. En nuestro caso, como se comentó inicialmente, sólo hemos implementado un método, *toMay,* para transformar una cadena de entrada en su correspondiente en mayúsculas. A continuación mostramos una captura de nuestra aplicación:

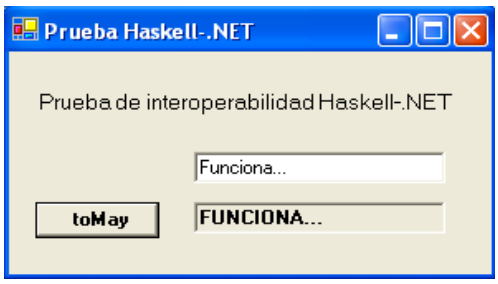

Figura 2. Nuestra aplicación de interoperabilidad.

4. Conclusiones

Son muchos los esfuerzos internacionales por desarrollar un marco global para la utilización de recursos informáticos, Microsoft tiene en .NET su apuesta más importante, que está revolucionando el mundo de los negocios y las comunicaciones a través de su nueva tecnología y los Servicios Web.

En el ámbito de la programación funcional, las iniciativas existentes para integrar este paradigma en el marco .NET, revelan el interés de la comunidad por converger a dicha plataforma. Es por ello que hemos encontrado este trabajo muy interesante, una manera de conectar ambos mundos, tan distintos en su esencia, pero capaces de aprovechar cada uno las ventajas del otro, mediante el uso de la tecnología de interoperabilidad existente.

Nuetra experiencia nos anima a seguir explorando las posibilidades que .NET ofrece al mundo de la programación funcional y viceversa. Nuestro próximo proyecto es conseguir interconectar la herramienta de terminación MU-TERM [Luc04] a través de una interfaz de usuario C# y explorar la posibilidad de utilizar ésta mediante el uso de Servicios Web basados en XML.

BIBLIOGRAFÍA

[COM] COM, Component Object Model
http://www.etse.urv.es/EngInf/assig/ens4/2004/net4a.pdf

[DLMMU04] F. Durán, S. Lucas, J. Meseguer, C. Marché, and X. Urbain. Proving Termination of Membership Equational Programs. In P. Sestoft and N. Heintze, editors, *Proc. of ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation, PEPM'04* , pages 147-158, ACM Press, New York, 2004.

[FPBBG03] J. Ferguson, B. Patterson, J. Beres, P. Boutquin, y M. Gupta. La biblia de C#. Anaya Multimedia, 2003.

[FLMP99] S.Finne, D. Leijen, E.Meijer, S. Peyton Jones. Calling hell from heaven and heaven from hell. In Proc. of the 4th ACM SIGPLAN International Conference on Functional Programming, ICFP'99, *Sigplan Notices* 34(9):114-125, 1999.

[Fin99] S. Finne. HaskellDirect User´s Manual. November 1999.

[HDi] H/Direct: supporting component programming in Haskell.
http://www.haskell.org/hdirect/design.html#toc3

[Hlu] B. Hludzinski. Understanding Interface Definition Language: A Developer's Survival Guide. http://www.microsoft.com/msj/0898/idl/idl.htm

[Hoa03] T. Hoare. The Verifying Compiler: A Grand Challenge for Computing Research. *Journal of the ACM*, 50(1):63-69, 2003.

[Luc04] S. Lucas. MU-TERM: A Tool for Proving Termination of Context-Sensitive Rewriting. In V. van Oostrom, editor, *Proc. of 15h International Conference on Rewriting Techniques and Applications, RTA'04*, LNCS 3091:200-209, Springer-Verlag, Berlin, 2004.
Available at http://www.dsic.upv.es/~slucas/csr/termination/muterm

[Ste04] P. Steele. 15 Seconds: COM Interop Exposed. 2004.
http://www.15seconds.com/issue/040721.htm

[Tro02] A. Troelsen. *COM and .NET Interoperability*. Apress. 2002.