Building .NET GUIs for Haskell applications

Beatriz Alarcón
DSIC, UPV, Camino de Vera s/n,
46022 Valencia, Spain
balarcon@dsic.upv.es

Salvador Lucas
DSIC, UPV, Camino de Vera
s/n, 46022 Valencia, Spain
slucas@dsic.upv.es

ABSTRACT

.NET is an emerging Microsoft's project which promotes a new framework for Software Development emphasizing the use of Internet resources and the interaction between components written in different programming languages. Whereas functional programming languages such as Haskell are well-suited for developing tools to analyze, verify and transform programs, typical Haskell compilers do not provide sophisticated capabilities such as support for XML-Web services, assisted GUI development, HTML processing, etc., which are frequent in most .NET development frameworks. We show how to integrate software components developed in a functional language as Haskell together with (graphic) components developed in C# or another .NET language. To achieve our objective we use the facilities offered by .NET to import COM components, on the one hand, and the technology developed to generate COM components from Haskell modules, on the other.

Keywords: COM, Haskell, Interoperability, .NET, Programming environments.

1 INTRODUCTION

International efforts to develop a global framework to use software resources have in Java and .NET their most well-known exponents. .NET is an emerging Microsoft's project which promotes a new framework for Software Development emphasizing the use of Internet resources and the interaction between components written in different programming languages [Cha02]. Within the .NET platform we can integrate already existing technologies and products as well as new elements. The XML project promoted by the WWW consortium[1] is also related to this effort through the use of XML to document programs in .NET, the support of Web services based on XML, etc.

The scientific communities that develop languages and declarative software technology are carrying out an important effort to make use of this kind of initiatives. Functional languages like Haskell[2] offer many programming features and resources which make them powerful tools for developing software projects and rapid prototypes. However, typical Haskell compilers (e.g. GHC, Hugs,...) do not provide visual tools for easily defining graphical user interfaces (GUIs), as, on the contrary, many other programming languages have. Although there are several libraries and systems which can be used to develop GUIs in Haskell (e.g., *wxHaskell*[3], *Gtk2Hs*[4], *HToolkit*[5], etc.), a Haskell programmer can waste too much time in giving form to his application if he make use of such tools due to the lack of a graphic assistant which makes easier the design of a GUI. With an Integrated Development Environment (IDE) like Visual Studio .NET, this is pretty simple. The support to define Web services offered by the .NET platform is a second aspect of Haskell applications for which we could argue similarly.

Of course, having graphic libraries for functional languages is very interesting and useful. Unfortunately, we can not affirm that such libraries (e.g., *wxHaskell*, which we have used to develop a large Haskell application like the termination tool MU-TERM [Luc04]) behaves like a completely stable and handy system (yet) since you have to make sure that you have the same version of the GHC compiler installed that requires the version of wxHaskell you want to use. The design, description, and use of forms and graphic controls is not very easy and it can take time to obtain what one is looking for. Moreover, it is necessary to get a grip on three basic concepts: widgets, layout and events.

This gave us a first motivation to start the research in this paper. Another (more general) motivation comes from the frequent need (in software development) of combining software pieces of code written in different programming languages. Of course, this is the well-known problem of *interoperability* of software components in sofware engineering and there are a number of *middleware* solutions available for dealing with this (also for Haskell applications, as we will see below). However, as far as we know, no attempt to *use* the .NET technology in practice (i.e., with a real Haskell application) has been reported yet. We have also tackled this task: In 1999, Finne et al. [FLMP99] explored the possibility of encapsulating Haskell programs like COM objects (Microsoft's *Component Object Model* [Rog97, COM04]). Why couldn't we take a step further and achieve our goal by means of COM and .NET interoperability? Microsoft has left opened the possibility of using already existing COM components in .NET; thus, a Windows programmer does not need to rewrite

[1] http://www.w3c.org
[2] http://www.haskell.org
[3] http://wxhaskell.sourceforge.net
[4] http://haskell.org/gtk2hs/
[5] http://htoolkit.sourceforge.net/

all his applications to run them under .NET. In our case, we show how to take advantage of this to pack Haskell programs as software components and integrate them into applications written in other languages, for example in C#, the most popular .NET language. Let's give a brief overview of our approach.

Our starting point is *HaskellDirect* (*HDirect* [Fin99, FLMP99, HDi99]) a framework for Haskell FFI (*Foreign Function Interface*) based on the standard IDL (*Interface Definition Language*) which allows to specify a programming interface in a programming language independent manner. There are many possibilities that HDirect offer to the programmer: Creating Haskell bindings to external libraries, creating external bindings to Haskell libraries, creating Haskell client interfaces to COM objects, and creating Haskell COM objects. In our case, starting from a Haskell component, we build a COM component which is encapsulated into a *Dynamic Link Library* (DLL), making it able to interoperate with Windows applications and, in particular with .NET applications. Our particular interest is furnishing Haskell applications with .NET GUIs, but most of the discussion is completely general and independent from this concrete goal. HDirect implements in Haskell all the required functionality to build a COM component and exempts the programmer from the knowledge of the COM specification since it is generated automatically. Next, we make use of the .NET facility to import COM components which can be used as external functions to implement the C# event handlers for the controls in the .NET GUI.

The paper is organised as follows: Section 2 briefly describes .NET graphic controls. Section 3 introduces a simple case study which we use to illustrate our development. Section 4 explains how to build a COM component from a Haskell module. Section 5 addresses the problem of its integration into .NET. Section 6 reports on the results obtained on a concrete (realistic) application of our technique. Section 7 displays our conclusions and lines of future research.

2 OVERVIEW OF .NET GRAPHIC CONTROLS

When a Windows programmer writes a .NET application (in, e.g., C#), he or she can take advantage of the System.Windows.Forms namespace, which provides a variety of control classes for developing rich user interfaces. Some controls are designed for data entry in the application (e.g., TextBox and ComboBox controls). Other controls display application data (e.g., Label and ListView controls). The namespace also provides controls for invoking commands within the application, such as the Button and MainMenu controls. In this paper we are specially interested in showing how Haskell applications can take advantage from .NET technology, specially from .NET GUIs. Thus, we only consider the information (or *data*) that graphic controls and Haskell components should (usually) *exchange*. Although other control properties (e.g., control labels, colors, etc.) could also be managed through Haskell components, we will not consider them in detail here; we center the attention on the non-graphic part of this information exchange . Extending the treatment of controls to achieve such more generality would be managed in a similar way, if necessary.

The hierarchy of .NET controls is very large. Here, according to [FPB$^+$03] we mention the most common controls (which are also the most frequently used, in our personal practice). We consider that these controls suffice for giving a complete account of the problems and solutions that any other control could rise and require to achieve our purpose.

The table in Figure 1 shows the Haskell-like data as could be considered to be managed by each .NET control. This table shows that with few simple Haskell datatypes can be managed all necessary information, regarding our main purpose of having the graphic part of the application developed in .NET (C#) and the 'logic' of the program written in Haskell.

3 A SIMPLE CASE STUDY

In order to discuss the techniques developed here, we use a simple case study. It includes a simple graphic interface to introduce and manipulate strings by means of simple transformations:

- converting the characters of the string into capital or small letters,
- removing spare blank spaces, and
- simple encryption (based on the well-known Caesar's method)

The length of each string is also stored (as an integer value). In order to highlight the role of Haskell as the language which actually implements the logic of the application, the use of C# here is strictly limited to provide a GUI, i.e., to ease the introduction and visualization of strings by means of graphic controls. The length of the *current* string is displayed in a read only text control. The different transformations are triggered by means of buttons. The current string is selected from a *ComboBox* which shows the strings introduced so far (see Figure 2).

In the Haskell part, we have the structures of functional data which are necessary to control the state of the system: we store each pair string-length in a list that is indexed by a integer that points out at the *current* position of the list (`Focus`):

WINDOWS FORMS	ASSOCIATED DATA
Button, GroupBox, Panel, Label, Splitter	-
CheckBox, RadioButton	Bool
ListBox	([Int],[String])
ComboBox	(Int,[String])
ListView	[[String]]
TrackBar, ProgressBar, NumericUpDown	Int
TextBox, RichTextBox	String
MainMenu, OpenFileDialog, SaveFileDialog, FolderBrowserDialog	-

Figure 1: .NET controls and data

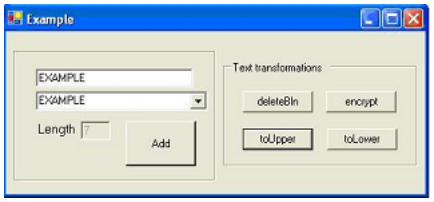

Figure 2: Simple example of interoperability

```
type Focus = Int
type Length = Int
data HL = H_L [(String, Length)]
          Focus deriving Show
```

The algebraic data type `HL` contains all necessary information to implement the required functionality explained above. The following mappings manipulate this data structure:

```
- Adds a new string and its length
addPair :: HL -> String -> HL
- Obtains the 'current' string
getString :: HL -> String
- Updates the 'current' string
writeString :: HL -> String -> HL
- Length of the 'current' string
getLength :: HL -> Int
- Sets the (index of) 'current' string
setFocus :: HL -> Int -> HL
```

The following mappings implement the transformations over strings.

```
toUpperCase :: String -> String
toLowerCase :: String -> String
deleteB :: String -> String
encrypt :: String -> String
```

Haskell files and other (IDL, C#, etc.) archives as explained below can be retrieved from `http://www.dsic.upv.es/~balarcon/example.zip`.

4 INTEROPERABILITY BY MEANS OF COM IN HASKELL

Microsoft's COM technology is used to create re-usable components (possibly written in different programming languages) and connect them together. In the following, we show how to use COM technology to connect Haskell with .NET components.

4.1 Haskell modules and COM components

A Haskell program that implements a COM component consists of four parts:

- The application code, written in Haskell by the programmer.
- An IDL specification establishing those Haskell functions which we want to make accessible through the DLL.
- A set of Haskell modules which are automatically generated from the IDL by the HDirect tool.
- A Haskell library module, Com, that exports all the functions needed to support COM objects in Haskell and a C library module that provides some run-Time support (RTS)

In the following sections we briefly describe and discuss these steps of the process.

4.2 The IDL of the Haskell component

IDL is a declarative language which is used to describe interfaces and classes disregarding any programming language [Hlu98]. An IDL specification describes the interface of a component.

The IDL code in Figure 3 is used in our case study. We have followed the example in [FLMP99], the indications of the manual of HDirect [Fin99] and the information about IDL [Hlu98]. We declare all (and only!) Haskell functions that we wish to have accessible from C# code together with their arguments and the type of the returned value.

Now we are going to describe the IDL code. This is useful to understand what we are going to obtain from COM [Rog97, Ste04, COM04]. On the basis of the IDL code, we are going to build the skeleton of the object that we want to encapsulate. For that purpose, we have a `library` (Example), an `interface` (Iexample) and a `class` (EXAMPLE).

```
[ uuid(35E80A56-3664-4d91-9C6C-3018496A8D61) ,
helpstring("Haskell COM component") ,
version(1.0) ]

library Example {

importlib("stdole32.tlb");

[ object,
 uuid(4DB0C045-CC9F-4607-B79A-26D27E0C1594)]

interface Iexample : IUnknown {

 HRESULT addPair([in,string]BSTR in);
 HRESULT getString([out,retval] BSTR *out);

 HRESULT getLength([out,retval] int *out);
 HRESULT setFocus([in] int in);
 HRESULT toUpperCase();
 HRESULT toLowerCase();
 HRESULT deleteB();
 HRESULT encrypt();

 };

[ object,
 uuid(49D98D24-DC88-4d24-8C5D-404FE510644D)]
coclass EXAMPLE {
  [default]interface Iexample;

 };

};
```

Figure 3: IDL code for the case study

A *type library* is a binary file that contains the same information that we could find in a C or C++ header file. It includes the names of the classes and the interfaces which are implemented in the server and the number and type of parameters for each method of their interfaces. Note that it also contains the GUID (Globally Unique IDentifiers), a very important part of the model of COM programming, for each class and interface. A GUID is a structure of 128 bits "statistically guaranteed" to be unique. In our case we have used the tool Create GUID (which is part of Visual Studio .NET) to generate them.

A *COM interface* is a collection of linked methods that perform a functionality. All are based on the *IUnknown* interface; each of them receives a unique *interface identifier* (IID).

A *COM class* is the implementation of one or more COM interfaces, while a COM object is an instance of a class. Each object has a class identifier (CLSID). CLSID and IID are subgroup of GUID.

The name of our interface is Iexample and inherits from IUnknown the use of methods QueryInterface, AddRef and Release. Inheritance from multiple interfaces is not allowed. The first attribute, `object`, which is locked up in brackets next to the GUID, identifies the interface as a COM interface. For each method in the interface, we specify the parameters with which the method will be called (from C#). The attribute `in` indicates that the parameter is used as an input given to the method (e.g., in addPair), `out` indicates output (e.g., in `getString`). The attribute `string` is used with parameters that are pointers to characters. The `retval` keyword indicates that the parameter must be interpreted as the returned value of the function. It must do it in this way, because the literal return of the method is a HRESULT type, which is used to give back the information of errors.

4.3 Encapsulating a Haskell component as a COM component

Once the IDL has been specified, the next step is to generate the *proxy* and the *skeleton* of our component. In order to generate those modules we use the following (HDirect) command:

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

This generates two Haskell files: EXAMPLE.hs and ExampleProxy.hs. The first one contains the *skeleton* of the methods that implement our component, that is, the Haskell structure for the methods declared in the IDL. The second one provides a *proxy* that adapts our methods behind an interface COM to make the communication possible.

Regarding the definition of the *skeleton*, HDirect accomplishes three fundamental tasks:

- To import the necessary Haskell modules to give support to the characteristics of the interface specified by the IDL.
- To introduce a `State` type to implement the (necessary) persistence of the functional data by means of a mutable variable that can be initialized, read and modified by means of the functions of the `IOExts` library of *GHC*[6].
- To include Haskell declarations corresponding to the functions defined in the IDL. Haskell functions will have an additional argument corresponding to the state of the application (that will be able to be read or modified) and a monadic return type `IO t` where `t` is the (non monadic) type returned as indicated in the IDL (`String` or `Integer`, in our case).

The following step is to fill up the *skeleton* with the Haskell code of our methods. In our case, the `HList` module contains the methods in pure functional code, so we will fill up the *skeleton* with the corresponding calls to methods and the operations to read and write, the state by means of `readIORef` and `writeIORef` (defined in `IOExts`). For instance, for `deleteB`, we have:

[6] Glasgow Haskell Compiler, `http://www.haskell.org/ghc`.

```
module EXAMPLE where (...)

import IOExts

-Pure Haskell Component
import qualified Hlist

data State = State(IORef HList.HL)
    ⋮
deleteB :: State
        -> Prelude.IO ()
deleteB (State st) = do
        hl <- readIORef st
       ;str' <- Prelude.return
                (HList.deleteB (HList.getString hl))
       ; writeIORef st(HList.writeString str' hl)
```

4.4 Creating a COM DLL from Haskell modules

The next step is to compile the two new files to generate the .hi and .o files and the *stubs* of the *proxy*:

```
ghc -c EXAMPLE.hs  ExampleProxy.hs
```

Now we have to decide how to encapsulate our component. HDirect provides solutions to build servers of internal processes (DLLs) or servers of external processes (EXEs). We have chosen to implement a DLL. Although it entails a bit more effort, the user benefits from a simpler use of the COM model, as the COM object is loaded without any intervention from the user.

In order to implement a DLL, the next step is to include the ComDllMain.lhs and dll_stub.c modules in the directory and compile them. Finally it is necessary to provide a Main module (required by GHC for descriptive purposes).

Once the module Main has been compiled, we build the type library (.tlb) using HDirect from the IDL and the proxy, generating *example.tlb*:

```
  ihc -s -fanon-typelib -v -c example.idl -o
ExampleProxy.hs -output-tlb=example.tlb
```

The type library is a resource that we must bind to our DLL. Resources are specified using a special and very simple text file, called resource script or .rc file. The file contains the specification of the resources that we want to include in the program or DLL (in our case the type library) for compiling it with the resource compiler. The resource compiler converts the file .rc into an object file (.o). The resource compiler is a GNU binary utility called *windres*. We use it along with *cygwin* to include *example.tlb* in our project. Now, we can build the DLL.

5 INTEGRATION OF COM INTO .NET

At this point, we must insert the COM DLL into our Visual Studio.NET project[7]. Having the DLL, it is necessary to register the generated component. The simplest way is using *regsvr32.exe*, in the command-prompt window. COM only uses a registry branch: HKEY_CLASSES_ROOT. Under it, we can find all the CLSIDs of the components installed in the system. A CLSID is contained in the registry as an alphanumeric string with the following format: {xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx}.

5.1 Using COM components in .NET applications

A .NET client cannot directly communicate with a COM component because the interfaces exposed by the COM component cannot be read from the .NET application. The data types, the mechanisms for managing errors, etc., are different for *managed* and *unmanaged* objects[8]. In order to simplify the interoperation between the components of .NET Framework and the unmanaged code, the CLR (Common Language Runtime) hides the differences between them both to clients and servers. This is achieved by means of a RCW (Runtime Callable Wrapper)(to understand the whole process see Figure 4) . The .NET SDK provides RCW to obtain it, thus a .NET application can see the unmanaged component as if it was managed. In .NET there are several ways to do this:

- Using the Type Library Importer utility (*Tlbimp.exe*), provided together with the .NET Framework.
- Making reference to the COM component directly from the C# application.

Tlbimp is a console application that converts the type definitions found in a COM type library into equivalent definitions in a .NET assembly. The assembly produced by the Tlbimp.exe tool is a standard .NET assembly that can be examined with *Ildasm.exe* (MSIL disassembler).

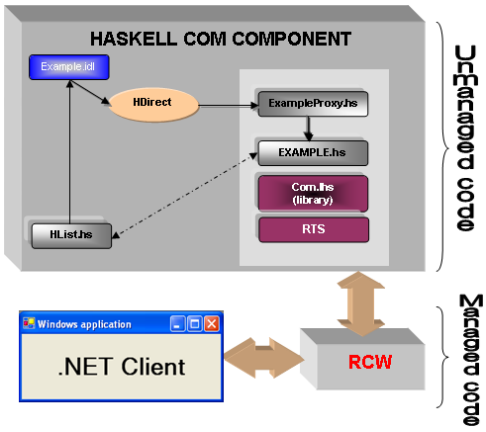

Figure 4: Interoperability with .NET from Haskell

After registering our DLL, we use *Tlbimp* and we run VS.NET. From our Windows application we click the

[7] We use Visual Studio.NET 2003.

[8] The .NET native CLR code is called 'managed', in contrast to any other machine-dependent code which is 'unmanaged' [Cha02].

right button on the *References* file in the VS *Solution Explorer*, we select *Add reference* and look for the assembly which we have just generated. Now it can be used exactly as any other .NET assembly: we just create an instance (denoted by `h`) of the appropriate class:

```
ExampleClass h = new ExampleClass();
```

Now we can access to Haskell functions as if they were C# functions (see Figure 5).

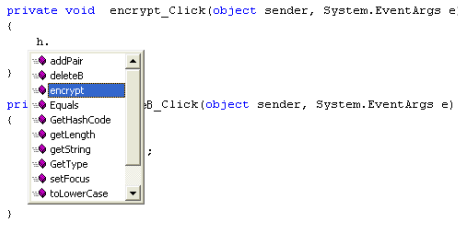

Figure 5: Haskell functions in C#

We can use them to program the event handlers on the GUI that we have developed.

6 A .NET VERSION OF MU-TERM

MU-TERM is a termination proof tool for (Context-Sensitive) Rewriting Systems. (Context-sensitive) Rewrite Systems are useful for describing semantic aspects of a number of programming languages (e.g., Maude, OBJ2, OBJ3, or CafeOBJ) and analyzing the computational properties of the corresponding programs, in particular termination (see [DLM+04, Luc01, Luc02]). The tool implements the generation of the appropriate orderings and transformations for proving termination. MU-TERM is written in Haskell and *wxHaskell* was used to develop the graphical user interface. The system consists of around 30 Haskell modules containing more than 5000 lines of code. We refer the reader to [Luc04] for more information about the use and functionality of the tool. Compiled versions and instructions for the installation are available on the MU-TERM WWW site.

We have developed a new (hybrid) version of MU-TERM which, having the same functionalities (implemented by the same Haskell modules), includes a GUI written in C# which replaces the old one. Let's take a look to the windows which constitute the GUI of MU-TERM (see Figures 6 and 7) and let's consider the corresponding controls. As it can be noticed, the controls to manage in the interface are *MenuFile*, *Button*, *ComboBox*, *CheckBox*, *TextBox*, *ListBox*, etc. In Section 2 we discussed them and their associated data. We have applied the process described in Sections 4 to 5 to MU-TERM and the obtained results were very satisfactory. The new version of MU-TERM is now composed of the same number of Haskell modules but the WinMuTerm.hs module, which contained about 1200 lines of code, has been replaced by a new module WinMuTermNET.hs, that contains less than 800 lines.

On the other hand, the C# part of the .NET version of MU-TERM (consisting of six new modules with about 2000 lines altogether, most of them generated automatically(!) by the graphic assistant) includes a new C# module WinMuTermNET.cs that implements the creation of the new user interface and manages the events transforming them in function calls to Haskell code by means of exchanges of strings and integers. This C# component uses the COM DLL generated from WinMuTermNET.hs (together with the other Haskell modules). The .NET version of MU-TERM is available on the MU-TERM WWW site.

7 CONCLUSIONS AND FUTURE WORK

We have shown how to integrate software components developed in Haskell together with (graphic) components developed in C#, or other .NET language. Our starting point is *HDirect* which permits to build a COM component from a Haskell module, and making it available as a COM DLL which can interoperate with .NET applications. We have shown the practicality of this approach by giving a new .NET GUI to a Haskell tool like MU-TERM. Other remarkable aspects are:

- it is a complete experience of 'weak' integration of software components written in a functional language like Haskell in a software development platform like .NET that still does not manage the inclusion of sources written directly in this language.
- it is a pioneer experience in the functional programming community, since MU-TERM is the first complex software written in Haskell that uses COM technology by means of HDirect.
- it is also a pioneer experience for the academic community interested in the interoperability of program analysis software tools, specially regarding tools for proving termination, where interoperability of different tools can be important [9].

In the world of functional languages, there are more or less complete approximations to .NET for the languages[10]. Regarding Haskell, a full-featured Haskell development environment has been recently implemented. It is called *Visual Haskell* [AM05]. Although it is an interesting contribution for the Haskell community, it does not treat the possibility of building graphic user interfaces for Haskell programs using the .NET resources. In their project they have also used HDirect, although they did not find it completely

[9] See, for instance, `http://www.lri.fr/~marche/termination-competition`.

[10] ML `http://www.cl.cam.ac.uk/Research/TSG/SMLNET` or Mondrian `http://www.mondrian-script.org`.

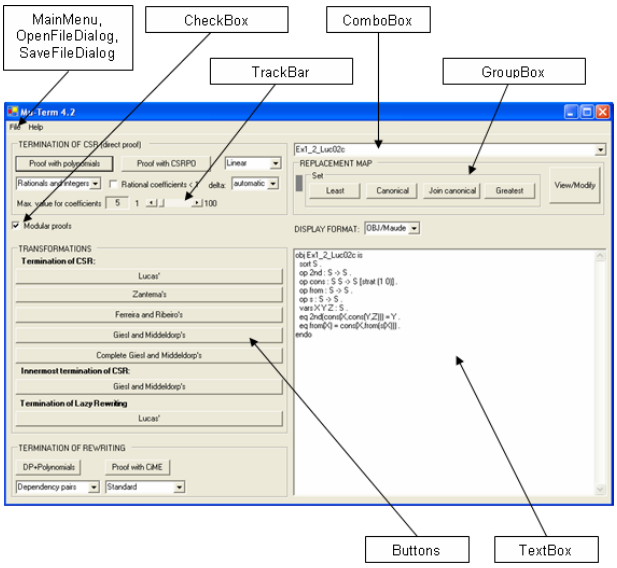

Figure 6: Principal window of MU-TERM

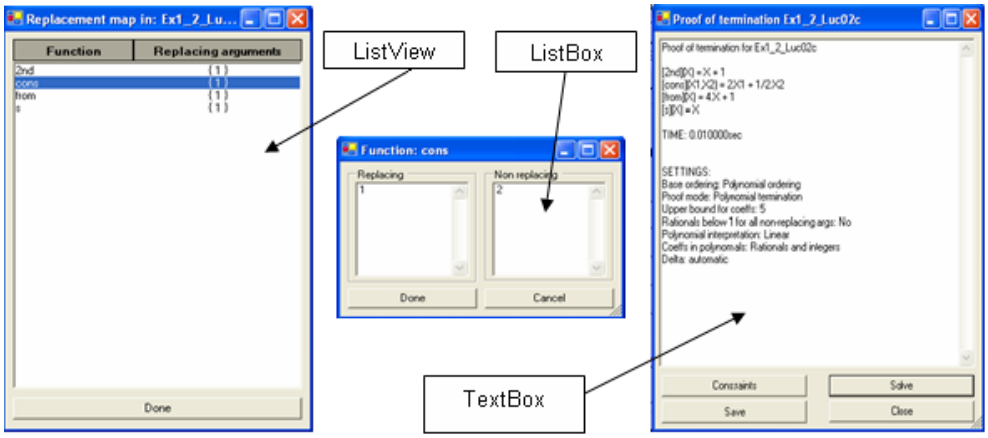

Figure 7: Rest of MU-TERM windows

appropriate for their purposes. This is also in contrast to ours: due to the simplicity of the information exchange between the C#-based user GUI and the core Haskell application, we find HDirect to be easy to use (although it took time to reach a sufficient *know-how*). For instance, HDirect limits the structures of Haskell data that are directly interchangeable by means of COM to strings and integers (of 32 bits). This can be a problem for most applications, but it is not problematic for developing GUIs, since the involved data types (see Section 2) are easily exchangeable in such format.

These initiatives to integrate functional languages into the .NET framework reveal the interest of the community to converge to this platform. Our experience is also encouraging. We plan to develop the theoretical aspects of our work, and also envisage possible extensions of this experience to other tools and programming languages in the future. In particular, we want to explore the use of the .NET facilities for using Web Services based on XML with these tools and programming languages. A first candidate, again, could be the termination tool MU-TERM.

ACKNOWLEDGEMENTS

Work partially supported by Spanish MEC grant SELF TIN 2004-07943-C04-02, Acción Integrada HU 2003-0003, and EU-India Cross-Cultural Dissemination project ALA/95/23/2003/077-054.

REFERENCES

[AM05] K. Angelov and S. Marlow. Visual Haskell. In *Proc. of Haskell Workshop, Haskell'05*, pages 5-16, ACM Press, 2005.

[Arc01] T. Archer. Inside C#. McGraw-Hill, 2001.

[Cha02] D. Chappell. Understanding .NET. Addison Wesley, 2002.

[COM04] COM, Component Object Model. `http://www.etse.urv.es/EngInf/assig/ens4/2004/net4a.pdf`

[DLM+04] 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.

[Fin99] S. Finne. HaskellDirect UserÂ«s Manual. November, 1999.

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

[FPB+03] J. Ferguson, B. Patterson, J. Beres, P. Boutquin, and M. Gupta. C#'s bible. Microsoft Press, 2003.

[HDi99] H/Direct: supporting component programming in Haskell. `http://www.haskell.org/hdirect/design.html#toc3`

[Hlu98] B. Hludzinski. Understanding Interface Definition Language: A Developers Survival Guide, 1998. `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.

[Luc01] S. Lucas. Termination of Rewriting With Strategy Annotations. In *Proc. of LPAR'01*, LNAI 2250:669-684, Springer-Verlag, Berlin, 2001.

[Luc02] S. Lucas. Context-sensitive rewriting strategies. *Information and Computation*, 178(1):293-343, 2002.

[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`.

[Rog97] D. Rogerson. Inside COM. Microsoft's Component Object Model. Microsoft Press, 1997.

[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.