The design of the wxOCaml library

Fabrice Le Fessant
INRIA & OCamlPro
fabrice.le_fessant@inria.fr

Abstract

WxWidgets is a well-known cross-platform library to build graphical user interfaces for desktop applications. It has bindings for many languages, among which the most famous is probably WxPython, but not for OCaml. Binding such a library is a huge — error-prone — work, because, to be useful, hundreds of C++ classes with tens of methods have to be accessible though the binding.

In this paper, we show how this problem was solved in the case of wxOCaml, providing an easy to use, easy to extend binding supporting multiple versions of WxWidgets.

1 Introduction

WxWidgets is a well-known cross-platform library to build graphical user interfaces. Many bindings for this library have been implemented for various languages, such as wxPython, wxHaskell, etc. Although a prototype was available for OCaml (wxCaml), the binding was very limited and not well-typed (all wxCaml widgets types were equivalent).

Although desktop interfaces are less used these days, they are still needed for many desktop applications, and especially for beginners who like to *see* their first programs running. GTK has a very good binding in OCaml (lablGTK), but is often not considered good enough for two reasons:

- The look-and-feel of GTK is not as native as other libraries, especially under Windows and Mac OS X.
- The binding uses all the power of OCaml type system: classes, objects, polymorphic variants, optional arguments are used everywhere. Consequently, it is hard to use for beginners, and sometimes even for experts, as error messages for objects are sometimes hard to understand, without a strict type-annotation discipline.

In this paper, we present a new binding for WxWidgets in OCaml, called wxOCaml, that provides all the advantages of WxWidgets over GTK, while still using only basic features of OCaml type system (no classes, overloading of virtual methods by simple records, etc.).

2 Design Choices

In WxOCaml, we wanted to map the complex C++ hierarchy of widgets' classes on a simplified version of the OCaml type system, i.e. without using OCaml classes, that would naturally fit them.

For that, we first define a type α `wx` to be used for all WxWidgets objects manipulated as OCaml values. For every C++ class `WxNAME`, we associate an abstract type `wxNAME_class` and every corresponding value in OCaml will have type `wxNAME = wxNAME_class wx`. All these definitions are gathered in a `WxClasses` module.

Here is the translation of the `minimal` example of WxWidgets:

```
open WxWidgets                    (* minimal example, menu + messageBox *)
open WxDefs                                    (* all symbolic constants *)
let _ =                  (* onInit is called from WxWidgets event loop *)
  let onInit (app : wxApp) =
    let frame_id = wxID () and quit_id = wxID() and about_id = wxID() in
    let frame = WxFrame.createAll None frame_id
        "Hello World"  (50, 50)  (450, 350) wxDEFAULT_FRAME_STYLE
    in
    WxFrame.setIcon frame (WxIcon.createFromXPM Sample_xpm.sample_xpm);
    WxMENU_BAR.(wxFrame frame [ "&File", [        (* Application menu *)
          Append(about_id, "&About");
          AppendSeparator();
          Append2(quit_id, "E&xit", "Exit from the application"); ]]);
    WxFrame.setStatusText frame  "Welcome to WxWidgets!" 0;
    WxEVENT_TABLE.(wxFrame frame () [              (* events handlers *)
        EVT_MENU(quit_id, (fun _ _ -> exit 0));
        EVT_MENU(about_id, (fun _ _ -> ignore_int (
              WxMisc.wxMessageBox "WxWidgets Hello World example."
                "About Hello World" (wxOK lor wxICON_INFORMATION)
            ))) ]);
    ignore_bool ( WxFrame.show frame );
    WxApp.setTopWindow (WxFrame.wxWindow frame)
  in
  wxMain onInit                           (* start WxWidgets event loop *)
```

Then, for every class `WxNAME`, we define a module `WxNAME` containing all the stubs of the methods that can be applied to values of that class. A static C++ method is just mapped to an equivalent OCaml function. A non-static C++ method is mapped to a function taking the object as first argument, and then other arguments of the method.

Note that the module does not only contain the stubs for methods of the class, but also for methods of all the ancestors of the class, since these methods can also be applied on an object of that class. However, although it might look expensive in code size to have so many stubs, it does not cost any extra runtime code, since the stubs are OCaml external primitives, and a stub for a method of an ancestor class directly calls the corresponding external primitive of ancestor class[1].

For inheritance, we provide *cast* functions in OCaml to cast an OCaml value matching a given C++ class to all of its ancestor classes. These functions are directly implemented by the identity function primitive of OCaml (`"%identity"`). We also provide a sub-module called `Unsafe` in every class, with cast functions to subclasses. These cast functions map to C++ `<dynamic_cast>` casts, and may raise an exception if the value is not of the expected subclass.

Finally, we also provide `properties` for each class, i.e. a set of methods, gathered as variants in a type, that change the state of the object and do not return any value. We provide a `set` function that takes a list of such properties, and call each method sequentially, and then returns the object. This function has proved very practical during the translation of C++ examples.

For example, here is the OCaml code for the binding of class `WxCommandEvent`:

```
open WxClasses       (* Specific methods of the WxCommandEvent class *)
external getInt : wxCommandEvent -> int = "wxCommandEvent_GetInt_c"
external getString: wxCommandEvent -> string = "wxCommandEvent_GetString_c"
...
(* Methods inherited from parents, if any *)
```

[1]This technique requires to dynamically type C++ objects, and to cast them before calling the method

```
external skip : wxCommandEvent -> bool  -> unit = "wxEvent_Skip_c"
external getTimestamp: wxCommandEvent -> int = "wxEvent_GetTimestamp_c"
...
(* Cast functions to parents (WxEvent and WxObject) *)
external wxEvent : wxCommandEvent -> wxEvent = "%identity"
external wxObject : wxCommandEvent -> wxObject = "%identity"
module Unsafe = struct    (* Cast functions to children, if any *)
  let wxCalendarEvent (x : wxCommandEvent) =
   (WxClasses.wxCast 76 43 x : wxCalendarEvent)
  ...
end
```

3 Implementation

The first implementation of wxOCaml was based on wxCaml, itself based on wxHaskell[2], itself using the manually written bindings of wxEiffel. While wxHaskell would generate Haskell stubs and classes from the include files describing the C prototypes of the wxEiffel stubs, wxCaml used `camlidl` to automatically generate OCaml stubs from these include files, modified to add some necessary IDL annotations.[2]

This solution had many drawbacks, for example:

- As a fork of wxHaskell, itself of wxEiffel, it would require to keep a close eye on these projects to follow their evolutions, that can conflict with wxOCaml evolution.
- A lot of C++/C stubs of wxEiffel in the include files are not correctly typed, or the types cannot be correctly mapped to OCaml types, requiring a lot of changes in these files, that would become difficult to synchronize with their original versions.
- Adding support for new classes/methods requires to manually write stubs to call the C++ method from a C function, and then to modify the include files to provide the prototype of the function, so that `camlidl` can generate the corresponding stubs.
- The (incomplete) information on the C++ class hierarchy was not used by `camlidl`, so that the only way to call ancestor methods from subclasses was to make all types equivalent, breaking type-safety. Although we experimented a partial solution by modifying `camlidl`, the results were considered not powerful enough.

The new implementation of wxOCaml does not share any code with other wxCaml/wxHaskell/wxEiffel projects. Instead, we implemented a new stub generator, with the following properties:

- It takes as input a list of files written in a specific DSL, close to C++ syntax, specifying both the class hierarchy and the prototypes of C++ methods. Adding support for a new method is usually as simple as adding the C++ prototype of that method, no C/C++ stub has to be written.
- It generates directly the `WxClasses` module containing all the class types, and for each class, the OCaml module and the C++ code. In the C++ code, only one C/C++ function per method is generated, with a C name (`extern "C"`) and C++ code, working correctly with the constraints of OCaml garbage collector.
- The DSL contains support for versioning, so that, depending on the version of WxWidgets installed (2.8 or 2.9), some stubs are generated or not.[3]

[2] wxEiffel provides C functions to call C++ methods, `camlidl` generates C functions to call these wxEiffel C functions from OCaml.

[3] If they are not, the function still exists, but raises an exception when called. This behavior can be altered by a configuration option at build time.

Here is the DSL file corresponding to the `wxCommandEvent` class:

```
class wxCommandEvent inherit wxEvent begin
  int      GetInt () const
  wxString GetString () const
  ...
 end
events wxCommandEvent [ (* The list of events using this type *)
   EVT_COMMAND_BUTTON_CLICKED
   EVT_COMMAND_CHECKBOX_CLICKED
   ...
]
```

4 Conclusion

The project started on March 28, 2013, the second implementation was started on April 9, 2013, and, on June 7, 2013, 1637 methods were implemented, corresponding to 96 WxWidgets C++ classes. Some non-trivial examples from WxWidgets were successfully translated to OCaml (`calendar` for date/time widgets, `keyboard` for mouse/key events, `drawing` for canvas manipulations, `stc` for the Scintilla editor, `wizard` for testing overloaded methods, etc.). The library is compatible with both WxWidgets 2.8 (current version) and 2.9 (next version).

We think this work is interesting for two main reasons:

- The library has an interesting set of widgets, providing a better look-and-feel on Windows and Mac OS X, while still being easy to use by beginners. Translation of C++ examples has shown that, although our design choices require to have explicit casts between classes (when used as an argument of a method in place of an ancestor), and full qualifications of function calls (`WxFrame.show frame` instead of `frame->show()`), the verbosity is not so high compared to C++ (especially thanks to local opening of modules using the `M.(...)` notation).
- Although our stub generator is specific to this library (it has support for specific features of WxWidgets, such as events), the approach can be used for other C++ libraries, to automatically generate stubs from a simple DSL description, close to C++ prototypes. Especially, we think that it would be much more difficult to generate a code equivalent to the code generated by our tool, using generic tools designed for other languages, such as SWIG[1].

We plan now to continue translating examples from WxWidgets from C++ to OCaml, adding support for the most commonly used widgets, and for the most useful methods of each widget, and providing a better support for versioning and configuration (i.e. missing dependencies causing missing widgets).

References

[1] D. M. Beazley. Swig: an easy to use tool for integrating scripting languages with c and c++. In *Proceedings of the 4th conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4*, TCLTK'96, pages 15–15, Berkeley, CA, USA, 1996. USENIX Association.

[2] D. Leijen. wxhaskell: a portable and concise gui library for haskell. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, Haskell '04, pages 57–68, New York, NY, USA, 2004. ACM.