

Automatically generated type-safe GTK+ binding for Dylan

Hannes Mehnert
hannes@mehnert.org
Dylan Hackers

ABSTRACT

We present an automated way to get language bindings for GTK+ for Dylan [2], an object-oriented functional programming language related to Lisp. Dylan supports multiple inheritance, polymorphism, multiple dispatch, keyword arguments, pattern-based syntax extension macros, and many other features. The generated binding is type-safe, no up- and downcasts are needed.

1. INTRODUCTION

Dylan was mainly influenced by Scheme and Lisp and adds an object system derived from CLOS. Dylan has an ALGOL-like syntax rather than a prefix syntax. It has a clear separation between compile and runtime, the compiler is not part of the runtime. By specification it does not need to have a read-eval-print-loop. The Open Dylan IDE includes a REPL for interactive development.

On current Desktop computers, Linux and other UNIX operating systems (like FreeBSD, OpenBSD, NetBSD) are getting more popular. This leads to the requirement to develop graphical user interfaces to this platform. These UNIX operating systems use the X.Org X Window System. Two major Desktop environments are KDE and Gnome. While KDE is written entirely in C++, which is hard to interface, Gnome is written in C, which can easily be interfaced with C-FFI (C foreign function interface ¹).

Developing a custom widget library is error-prone, a lot of work and will not easily have the same look and feel like a native application. Thus automatically generating a binding for an existing widget library like GTK+ is a decent solution. This paper will show how we succeeded to automatically generate a type-safe GTK+ binding for Open Dylan (former Functional Developer, DylanWorks, Harlequin Dylan). The purpose was to develop a DUIM (Dylan User

Interface Manager ²) backend for UNIX operating systems. DUIM is a GUI abstraction written by Scott McKay, who previously developed CLIM (Common Lisp Interface Manager).

Section 2 will introduce the GTK+ object system and what was needed to generate a binding for it. Section 3 will look at related work. Section 4 gives an overview what we achieved.

2. SOLUTION

2.1 Overview of GTK

‘GTK+ is a highly usable, feature rich toolkit for creating graphical user interfaces which boasts cross platform compatibility and an easy to use API.’ ³

GTK+ consists of multiple modules; ‘GLib, a low-level core library which forms the basis of GTK+. It provides data structure handling for C, portability wrappers and interfaces for such run-time functionality as an event loop, threads, dynamic loading and an object system’ ⁴. Pango, which lays out and renders text, and Cairo, a library for 2D graphics, among others.

GLib contains an object system with single inheritance and interfaces. It also specifies properties to store data in an object. There are universal getters and setters for properties. It is event driven and has the concept of signals. Whenever something changes, all connected clients get a signal. A client is a previously registered callback (a closure, each signal specifies the number of arguments the closure gets).

2.1.1 GTK+ Type system and introspection

The type system used is implemented in GLib. Each **GTypeInstance** (abstract), the top object of the type hierarchy (the non-abstract top is **GObject**), instance has a runtime type, a field in its struct. The type system provides only single inheritance, thus the type hierarchy is a tree. There are also interfaces, an abstract types used to specify an interface ⁵. For example a **GtkTreeStorage** implements the **GtkTreeModel** interface. Most **GObject** subtypes are opaque, access to member variables is wrapped into a method, which possibly does side effects, like sending

¹http://www.opendylan.org/documentation/opendylan/interop1/intero_4.htm

²<http://www.opendylan.org/documentation/opendylan/dguide/index.htm>

³<http://www.gtk.org/>

⁴<http://www.gtk.org/overview.html>

⁵generic sense of term

a signal, etc. Direct member access should not be used by developers.

The single inheritance type hierarchy can be introspected with the **GObject** method `GType g_type_parent(GType type)`, providing supertypes of a given type. Also, implemented interfaces can be extracted for a given type by the **GObject** method `GType* g_type_interfaces(GType type, guint* n_interfaces)`.

Since Dylan provides a type system with multiple inheritance, we modeled interfaces as “mixin” classes, resulting in a graph class hierarchy (in contrast to the tree of GTK+). **gobject2-tool**, a small tool written in C using GLib extracts from all GTK+ types their superclasses and interfaces. This generates an input file for generating C-FFI definitions.

Whenever a runtime GTK+ object enters the Dylan world, the GTK+ introspection features are used to lookup which actual type this pointer has. Then the respective Dylan class is instantiated. Thus, when `gtk.button_new` returns a pointer of **GtkWidget** in the GTK+ documentation, the real introspected type is a **GtkButton**, so a Dylan object of type `<GtkButton>` is instantiated.

This is done in the following Dylan method

```
define method make
  (type :: subclass(<GTypeInstance>), #rest args,
   #key address, #all-keys)
=> (result :: <GTypeInstance>)
  if(address)
    let instance = next-method(<GTypeInstance>,
                               address: address);
    if (~ null-pointer?(instance))
      let g-type = g-type-from-instance(instance);
      let dylan-type = find-gtype(g-type);
      unless (dylan-type)
        error("Unknown GType %= encountered.",
              as(<byte-string>, g-type-name(g-type)));
      end;
      let result =
        next-method(dylan-type, address: address);
      g-object-ref-sink(result);
      finalize-when-unreachable(result);
      result;
    else
      next-method();
    end
  else
    error("Can't instantiate GTypeInstance %=",
          type.debug-name);
  end if;
end method make;
```

First an object of type `<GTypeInstance>`, named **instance**, is created. The only purpose of this object is to lookup the real type of the given pointer with `g-type-from-instance`, a call of the C-macro `G_TYPE_FROM_INSTANCE` provided by GTK+. `find-gtype` translates the given GTK+ **g-type** to a Dylan type, search-

ing through all subclasses of `<GTypeInstance>`, comparing its name with the **g-type** name. Once found, the mapping from **g-type** to the respective Dylan type is cached in a hashtable.

To not leak memory, `g-object-ref-sink` is called and once the object is unreachable, it is cleaned up by the finalizer by calling `g-object-unref`.

Since the GTK+ type hierarchy is integrated into the Dylan type system, there is no need for upcasts. The compiler knows that a **GtkButton** is a superclass of a **GtkWindow**, and when `gtk.widget_show` is called on a **GtkButton**, that this is a proper argument for the method.

2.1.2 GTK public interface

Melange, a C interface generator, initially developed for Gwydion Dylan, was extended with a C-FFI backend for Open Dylan. Melange is able to parse C code and generating Dylan interface definitions for the given code. The previously generated type hierarchy is used as input for melange, as well as `gtk/gtk.h`.

Melange parses `gtk/gtk.h` and all included header files. For each constant, method and struct a foreign function definition is generated. When a type definition is encountered, the generated type hierarchy is used to output a proper Dylan type definition (using `define C-subtype`) with supertypes and implemented interfaces.

The GTK+ binding consists of more than 6500 function, 550 C-struct, 200 subtype and 4300 constant definitions. This is all done automatically, no manual intervention is needed.

GTK+ has been interfaced by several programming languages. Since interfacing of methods with variable number of arguments in C is not easy, GTK+ contains for all methods with variable number of arguments always methods with a static number of arguments. So we don't have to interface the methods with varargs, which currently can't be expressed in Dylans C-FFI.

2.1.3 Properties

GLib introduces properties for nearly all data slots of their objects. Those properties are set via the function, `void g_object_set_property(GObject* object, const gchar* property_name, const GValue* value)`. The function `g_object_get_property` (same signature as set) is provided by the GTK+ API to get a property.

To access the properties in Dylan, we extended our previously developed **gobject2-tool**. It now also generates lists of properties of all types in a simple syntax, for example `define property-getter logo :: <GdkPixbuf> on <GtkAboutDialog> end;`, which is a call to the following Dylan macro:

```
define macro property-getter-definer
  { define property-getter ?name :: ?type:name
    on ?class:name end }
=>
  { define method "@ " ## ?name (object :: ?class)
```

```

=> (res)
  with-stack-structure (foo :: <GValue>)
    g-object-get-property(object, ?"name", foo);
    g-value-to-dylan(foo);
  end;
end; }
end;

```

This is the macro definition. The left-hand side (surrounded by curly braces) matches the caller, and binds **?name** to **logo**, **type** to **<GdkPixbuf>** and **?class** to **<GtkAboutDialog>**. The right-hand side (after **=>**), also surrounded with curly braces) is the macro expansion. A method named **@logo** is defined (**##** is the splicing operation), which first calls **g-object-get-property** (**?"variable"** is conversion to a string of the given variable). Afterwards **g-value-to-dylan** is called, which translates the boxed **GValue** to a Dylan object (removing the boxing) with the value of the **GValue** and the type of the **GValue**.

For setting properties, a similar macro **property-setter-definer** is used.

We used **@** as a magic symbol to differentiate from normal functions and GTK+ properties. Thus, **gtkaboutdialog.@logo** gets the logo property of the **gtkaboutdialog**, **gtkaboutdialog.@logo := mylogo** sets the logo property to **mylogo**.

2.1.4 Callbacks for signals

GLib is event driven and has the concept of signals. Each time something changes, a signal is produced and sent to all connected callbacks of the specific signal at the instance. A callback is a closure which gets a specific set of arguments, different from signal to signal.

To interoperate with this structure, we needed a way to register Dylan methods as callback for GTK+ signals.

The major problem is marshalling and unmarshalling of Dylan objects to C. Section 2.1.1 showed the conversion of C pointers to Dylan objects. To work with callbacks, the other way, converting Dylan objects to C, is also needed.

Each time a signal is connected, a **GClosure** object is instantiated, and its meta marshaller is set to **dylan-meta-marshaller**, a custom marshaller which loops over the parameter list and instantiates proper Dylan objects from the C pointers.

```

define function dylan-meta-marshaller
  (closure :: <GClosure>,
   return-value :: <GValue>,
   n-param-values :: <integer>,
   param-values :: <GValue>,
   invocation-hint :: <gpointer>,
   marshal-data :: <gpointer>)

```

```

  let values = #();
  for(i from 0 below n-param-values)

```

```

    let value = make-c-pointer
      (<GValue>,
       primitive-machine-word-add
        (primitive-cast-pointer-as-raw
         (primitive-unwrap-c-pointer(param-values))),
        integer-as-raw
         (i * sizeof-gvalue())),
      #[]);
    values := pair(g-value-to-dylan(value), values);
  end for;
  values := reverse!(values);

```

```

*holding-gdk-lock* := *holding-gdk-lock* + 1;
let res =
  apply(import-c-dylan-object
         (c-type-cast(<C-dylan-object>,
                     marshal-data)),
         values);
*holding-gdk-lock* := *holding-gdk-lock* - 1;

```

```

if(return-value ~= null-pointer(<gvalue>))
  select(g-value-type(return-value))
    $G-TYPE-BOOLEAN => g-value-set-boolean
      (return-value,
       if(res) 1 else 0 end);
    $G-TYPE-NONE, $G-TYPE-INVALID => ;
    otherwise =>
      error("Unsupported GType: %=",
            g-value-type(return-value));
  end select;
end if;
end;

```

This is the most unsafe code of our binding. Since we can't iterate over a collection of C **GValues**, we do it manually with pointer arithmetic, instantiating **C-pointer** of type **GValue**. Each parameter passed is translated by **g-value-to-dylan** to a Dylan object (first code block). And afterwards the given Dylan closure **marshal-data** is called with the Dylan arguments (second code block). Afterwards the result value is marshalled back into a C object (last code block).

To call the **dylan-meta-marshaller** from C, a C entry point for this method has to be defined. This is done by using **C-callable-wrapper**, which is part of Dylans C-FFI.

```

define C-callable-wrapper _dylan-meta-marshaller
  of dylan-meta-marshaller
    parameter closure      :: <GClosure>;
    parameter return-value :: <GValue>;
    parameter n-param-values :: <guint>;
    parameter param-values :: <GValue>;
    parameter invocation-hint :: <gpointer>;
    parameter marshal-data  :: <gpointer>;
    c-name: "foo";
  end;

```

To connect a signal to a **<GObject>** using a callback, **g-signal-connect** is used. This uses

`_dylan-meta-marshaller` to marshal the arguments of the callback. The callback is passed as user-data C-pointer (`export-c-dylan-object`) to the meta-marshaller, which translates it back to a Dylan method and applies the given arguments to the method (see above).

```
define function g-signal-connect
  (instance :: <GObject>,
   signal :: <string>,
   function :: <function>,
   #key run-after? :: <boolean>)

  register-c-dylan-object(function);
  let closure = g-closure-new-simple
    (sizeof-gclosure(),
     null-pointer(<gpointer>));
  g-closure-set-meta-marshal
    (closure, export-c-dylan-object(function),
     _dylan-meta-marshaller);
  g-signal-connect-closure
    (instance, signal,
     closure, if(run-after?) 1 else 0 end)
end function g-signal-connect;
```

2.1.5 Multithreading

GTK+ should only be called from one thread, all GTK+ API calls should be surrounded by a call to `gdk-threads-enter` and `gdk-threads-leave`. There must be no nested calls to `gdk-threads-enter`.

We defined a macro, `with-gdk-lock`, which calls `gdk-threads-enter` at the beginning and `gdk-threads-leave` in the cleanup block, increments and decrements a variable `*holding-gdk-lock*` for the number of nested invocations of that macro. Thus, a developer does not need to care about `gdk-threads-enter`, but can simply use the around-macro.

```
define macro with-gdk-lock
  { with-gdk-lock ?body end }
=>
  { block()
    *holding-gdk-lock* > 0 | gdk-threads-enter();
    *holding-gdk-lock* := *holding-gdk-lock* + 1;
    ?body
  cleanup
    *holding-gdk-lock* := *holding-gdk-lock* - 1;
    *holding-gdk-lock* > 0 | gdk-threads-leave();
  end }
end;
```

When a callback of a signal is invoked, there is no need to call `gdk-threads-enter`, because GTK+ already takes care of it. This is the reason why we simply increment `*holding-gdk-lock*` in the `dylan-meta-marshaller` before we call the Dylan closure, and decrement it afterwards.

2.2 Toolchain

To summarize, the following tools were developed for the GTK+ binding:

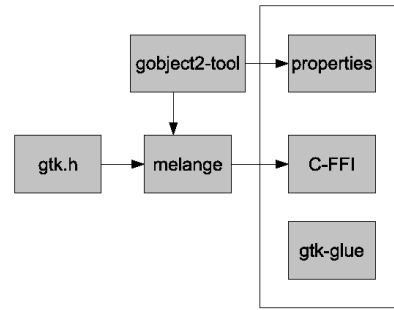


Figure 1: Interaction of tools, box on the right: the GTK+ binding

- `gobject2-tool`: C/GLib, introspects GLib for properties, supertypes and interfaces of GTK+ types
- `melange`: Dylan, parses `gtk/gtk.h` and generates FFI definitions, respecting type hierarchy
- `gtk-gluе`: Dylan, defines properties of GTK+ types, make and meta-marshaller trampoline, initialize-gtk

2.3 GTK+ hello-world

A GTK+ hello-world application written in C is shown. `G_OBJECT`, `G_CALLBACK`, `GTK_CONTAINER` are macros which cast the type.

```
static void
on_destroy (GtkWidget* widget, gpointer data)
{
  gtk_main_quit();
}

static void
on_click (GtkButton* button, gpointer data)
{
  gtk_button_set_label(button, "Hi");
}

int
main (int argc, char* argv[])
{
  GtkWidget* window;
  GtkWidget* button;

  gtk_init(&argc, &argv);
  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  g_signal_connect(G_OBJECT (window), "destroy",
                  G_CALLBACK (on_destroy), NULL);
  button = gtk_button_new_with_label("Hello, World");
  g_signal_connect(G_OBJECT (button), "clicked",
                  G_CALLBACK (on_click), NULL);
  gtk_container_add(GTK_CONTAINER (window), button);
  gtk_widget_show(button);
  gtk_widget_show(window);
  gtk_main();
  return 0;
}
```

This displays a widget with a button, which label is “Hello, World”, and once clicked changed to “Hi”.

The same functionality implemented with our Dylan GTK+ binding is shown in the following fragment.

```
define method gtk-hello-world () => ()
  initialize-gtk();
  let window = gtk-window-new($GTK-WINDOW-TOPLEVEL);
  let button
    = gtk-button-new-with-label("Hello, world!");
  gtk-container-add(window, button);
  g-signal-connect(button, "clicked",
    method(#rest args) button.@label := "Hi"; end);
  g-signal-connect(window, "destroy",
    method(#rest args) gtk-main-quit() end);
  gtk-widget-show(button);
  gtk-widget-show(window);
  gtk-main();
end method main;
```

This code is more concise for two reasons: anonymous methods and no need for type casts.

2.4 DUIM GTK+ backend

DUIM has a working Windows backend, as well as unfinished/non-working Motif and GTK-1 backends. After we developed a type-safe GTK+ interface, we started working on a GTK+ backend for DUIM, starting from the GTK-1 backend (which used a GTK interface which didn't reflect the type hierarchy and thus needed pointer casts).

This is currently not feature-complete, pixmaps, colors, etc. are missing. It currently consists of 8000 lines of code, but a big part are currently dead code.

Simple DUIM applications run without problems, the Open Dylan IDE has not yet been ported to the GTK+ backend. Network night vision [1], a graphical network sniffer written in Dylan, is working with DUIM using the GTK+ backend.

Screenshot at:

<http://www.opendylan.org/~hannes/gui-sniffer.png>

3. RELATED WORK

GTK+ has bindings for multiple programming languages (Perl, Python, Ruby, Java, Lisp, etc.), some are automatically generated, some are hand-written and need a big amount of work when updating to a newer version. Several bindings also treat a GTK+ object as a raw pointer, not allowing the automated type checks we have implemented.

4. CONCLUSION

We succeeded in automatically generating a type-safe GTK+ binding for Dylan. Thus, we won't pass invalid pointers to GTK+. It is also multi-threaded safe, as long as the developer uses `with-gdk-lock`. And the binding does not leak memory or interfere with our garbage collector.

Upgrading to a new GTK+ version takes little time, only running the complete chain of tools once. The GTK+

binding was successfully tested on Linux, Windows and FreeBSD.

The full source code is available at <http://www.opendylan.org/>. The code was developed with Andreas Bogk. Also thanks to Sven Neumann for deep knowledge of GTK+.

5. REFERENCES

- [1] MEHNERT, H., AND BOGK, A. A domain-specific language for manipulation of binary data in dylan.
- [2] SHALIT, A. *The Dylan Reference Manual*. Apple Press, 1998.