Biglook: a Widget Library for the Scheme Programming Language

Erick Gallesio
Université de Nice – Sophia Antipolis
950 route des Colles, B.P. 145
F-06903 Sophia Antipolis, Cedex
`Erick.Gallesio@unice.fr`

Manuel Serrano
Inria Sophia Antipolis
2004 route des Lucioles – B.P. 93
F-06902 Sophia-Antipolis, Cedex
`Manuel.Serrano@inria.fr`

Abstract

Biglook is an Object Oriented Scheme library for constructing GUIs. It uses classes of a CLOS-like object layer to represent widgets and Scheme closures to handle events. Combining functional and object-oriented programming styles yields an original application programming interface that advocates a strict separation between the implementation of the graphical interfaces and the user-associated commands, enabling compact source code.

The Biglook implementation separates the Scheme programming interface and the native back-end. This permits different ports for Biglook. The current version uses GTK+ and Swing graphical toolkits, while the previous release used Tk. It is available at: `http://kaolin.unice.fr/Biglook`.

Introduction

We have studied the problem of constructing GUI in functional languages by designing a widget library for the Scheme programming language, called Biglook. In this paper we focus on how to apply the functional style to GUIs programming.

Biglook's primary use is to implement graphical applications (*e.g.*, `xload`, editors *à la* Emacs, browser *à la* Netscape, programming tools such as `kbrowse`, our graphical Scheme code browser). Figure 1 presents two screen shots of Biglook applications: *i)* `kbrowse` on the left and *ii)* the Biglook dock applications, *à la* NextStep, on the right. These Biglook applications are used on a daily basis.

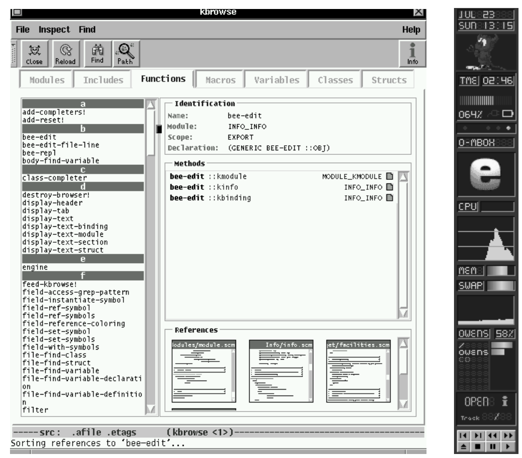

Figure 1: Two Biglook applications: a code browser on the left, "dock" applications on the right

By contrast to previous work, no attempt has been made to make that library familiar to programmers used to imperative or purely object oriented programming style. On the contrary, our library introduces an original application programming interface (API) that benefits from the high level constructions of the extended Scheme implementation named Bigloo [25] which is open source and freely available since 1992. The main Bigloo component is an optimizing compiler that delivers small and efficient applications for the Unix™ operating system. Bigloo is able to produce native code (via C) and JVM bytecode. Currently Biglook uses GTK+ [20] associated with the Bigloo C back-end and Swing [29] with the Bigloo JVM back-end. The previous release of Biglook [12] used Tk [19].

Bigloo implements an object layer inspired by CLOS [1]. It is a class-based model where methods override generic functions and are not declared inside classes as in Smalltalk [13], O'Caml [22] or Java [14].

Biglook is implemented as a wrapping layer on top of native widget libraries (that we name henceforth the *back-end*). This software architecture saves the effort of implementing low-level constructions (pixel switching, clipping, event handling and so on) allowing to focus on the Scheme implementation of new features.

When designing the Biglook API, we always had to decide which model to choose: the functional model or the object model. We think that these two models are not contradictory but complementary. For instance, if the widget hierarchy naturally fits a class hierarchy, user call-backs are naturally implemented by the means of Scheme closures.

In Section 1 we briefly present the Bigloo system emphasizing its module system and its object layer. This section is required for readers unfamiliar with the CLOS model and it also serves as an introduction to *virtual slots*. *Virtual slots* are a new Bigloo construction that is required in order to separate the Biglook API made of classes and the native back-end. They are presented in Section 2. In Section 3 we present the Biglook library. We start showing a simple Biglook application and its associated source code. Then, we detail the Biglook programming principles (widgets creation, event handling, etc.) motivating our design orientations by programming language considerations. In this section we are conducted to compare the functional programming style and the object oriented one. In Section 4 we present the Biglook implementation. At last, in Section 5 we present a comparison with related work.

1 Bigloo

Bigloo is an open implementation of the Scheme programming language. From the beginning, the aim was to propose a realistic and pragmatic alternative to the strict Scheme [17]. Bigloo does not implement "all" of Scheme; for example, the execution of tail-recursion may allocate memory. On the other hand, Bigloo implements numerous extensions: support for lexical and syntactic analysis, pattern matching, an exception mechanism, a foreign interface, an object layer and a module language. In this Section, we present Bigloo's modules and its object model; that is, class declarations and generic functions. *Virtual slots*, which are heavily used in Biglook, are presented in Section 2.

1.1 Modules

Bigloo modules have two basic purposes: one is to allow separate compilation and the second is to increase the number of errors that can be detected by the compiler. Bigloo modules are simple and they have been designed with the concern of an easy implementation.

A module is a compilation unit for Bigloo. It is represented by one or more files and has the following syntax:

```
(module module-name
   (import import+)*
   (export export+)*
   (static static+)*
   (library static+)*
)
opt-body
```

Import clauses are used to import bindings in the module. In order to import, one just needs to state the identifier to be imported and its source module. Note that a shorthand exists to import all the bindings of a module.

Export and *static* clauses play a close role. They point out to the compiler that the module implements some bindings and distinguish those that can be used within other modules (they are exported) and those that cannot (they are static). These clauses do not contain identifiers but prototypes. It is then possible to export variables (mutable bindings) or functions (read-only bindings). Static clauses are optional (the bindings of a module which are not referenced in a clause are, by default, static).

Library clauses enable programs to use Bigloo libraries. A Bigloo library is merely a collection of pre-compiled modules. Using a library is equivalent to *importing* all the modules composing the library. Detailed information on Bigloo modules may be found in the two papers [25, 26].

1.2 Object layer

In this paper we assume a CLOS-like object model with single inheritance and single dispatch. The object layer implemented in Bigloo is a restricted version of CLOS [1] inspired, to a great extent, by MEROON [21].

1.2.1 Class declarations

Classes can be declared static or exported. It is then possible to make a declaration accessible from another module or to limit its scope to one module. The abbreviated syntax of a class declaration is:

```
(class class-id::super-class-id opt-init opt-slots)
```

A class can inherit from a single super class. Classes with no specified super class inherit from the `object` class. The type associated with a subclass is a subtype of the type of the super class.

A class may be provided with an initialization function (*opt-init*) that is automatically called each time an instance of *class-id* is created. Initialization functions accept one argument, the created instance.

A slot may be typed (with the annotation `::type-id`) and may have a default value (`default` option). Here are some possible declarations for the traditional `point` and `point-3d`:

```
(module module-points
   (export (class point
               (point-init)
               (x::double (default 0.0))
               (y::double (default 0.0)))
            (class point-3d::point
               (z::double (default 0.0)))))

(define point-init
   (let ((count 0))
      (lambda (obj::point)
         (set! count (+ 1 count))
         (print "# of points: " count))))
```

1.2.2 Instances

When declaring a class *cla*, Bigloo automatically generates the predicate `cla?`, an allocator `instantiate::cla`, an instance cloner `duplicate::cla`, accessors (e.g., `cla-x` for a slot `x`), modifiers (e.g., `cla-x-set!` for a slot `x`) and an abbreviated special access form, `with-access::cla`, to allow accessing and writing slots simply by using their name. Here is how to allocate and access an instance of *point-3d*:

```
(let ((p (instantiate::point-3d
            (y -3.4)
            (x 1.0))))
   ;; The initialization value of a slot can be omitted
   ;; from the arguments list if it has a default value;
   ;; this is the case for the slot z of class point-3d.
   (with-access::point-3d p (x y z)
      (sqrt (+ (sqr x) (sqr y) (sqr z)))))
```

The `instantiate` and `with-access` special forms are implemented by the means of macros that statically resolve the keyword parameters (such as `x`, `y` and `z`). For instance, the above example is expanded into:

```
(let ((p (make-point-3d 1.0 -3.4 0.0)))
   (sqrt (+ (sqr (point-3d-x p))
            (sqr (point-3d-y p))
            (sqr (point-3d-z p)))))
```

As we can see, since macros are expanded at compile-time, there is no run-time penalty associated with keyword parameters.

1.2.3 Generic functions and methods

Generic function declarations are function declarations annotated by the `generic` keyword. They can be exported, which means that they can be used from other modules and that methods can be added to those functions from other modules. They can also be static, that is, not accessible from within other modules, hich means that no other modules can add methods. The CLOS model fits harmoniously with the traditional functional programming style because a function can be thought as a generic function overridden with exactly one method. The syntax to define a generic function is similar to an ordinary function definition:

(define-generic (*fun*::*type* *arg*::*class* ...) *opt-body*)

Generic functions must have at least one argument as this will be used to solve the *dynamic dispatch* of methods. This argument is of a type T and it is impossible to override generic functions with methods whose first argument is not of a subtype of T. Methods are declared by the following syntactic form:

(define-method (*fun*::*type* *arg*::*class* ...) *body*)

Methods override generic function definitions. When a generic function is called, the most specific applicable method, that is the method defined for the closest dynamic type of the instance, is dynamically selected. A method may explicitly invoke the next most specific method overriding the generic function definition for one of its super classes (a class has only one *direct* super class but several *indirect* super classes) by the means of the `(call-next-method)` form. It calls the method that should have been used if the current method had not been defined.

Here is an example of a generic function that illustrates the use of the Bigloo object layer. We are presenting a function that prints the value of the slots of the `point` and `point-3d` instances. This generic function is named `show`:

```
(define-generic (show o::point))
```

Then, the generic function is overridden with a method for classes `point` and `point-3d`.

```
(define-method (show o::point)
   (with-access::point o (x y)
      (print "x=" x)
      (print "y=" y)))

(define-method (show o::point-3d)
   (with-access::point-3d o (z)
      (call-next-method)
      (print "z=" z)))
```

Hereafter is an example of a call to the `show` generic function.

```
(let ((p (instantiate::point-3d
            (x 10)
            (y 20)
            (z 465))))
   (show p)) ⊢ x=10 y=20 z=465
```

2 Virtual slots

Bigloo supports two kind of instance slots: regular slots that have already been described in Section 1.2.1, and *virtual slots* that enable several views of a single data. As we will see in Section 4.2, *virtual slots* are at the heart of the Biglook implementation. They are mandatory to present Biglook to the user as a class based API. In particular, wrapping native widgets (such as GTK+ or Swing) for the Bigloo object model requires virtual slots.

Using virtual slots gives the illusion of accessing the slots of a class instance but instead, Scheme functions are called. As we have seen in Section 1.2.2, the compiler automatically defines *getters* and *setters* that access the various values embedded in the instances regular slots. Accessing virtual slots is syntactically identical to accessing plain slots, but virtual slots differ in the following way:

- their getters and setters are not generated by the compiler. They are defined by the user, in the class definition, using the class slot options: `get` and `set`.
- they are not allocated into memory.

For instance, let us consider a possible rectangle class implementation. An instance of `rect` is characterized by its origin (`x0`, `y0`) and either its upper right point (`x1`, `y1`) or its dimension (`width`, `height`). In the following class definition, the `width` and `height` slots are virtual.

```
(class rect
  x0 y0 x1 y1
  (width (get (lambda (o)
                (with-access::rect o (x0 x1)
                   (- x1 x0))))
         (set (lambda (o v)
                (with-access::rect o (x0 x1)
                   (set! x1 (+ x0 v))))))
  (height (get (lambda (o)
                 (with-access::rect o (y0 y1)
                    (- y1 y0))))
          (set (lambda (o v)
                 (with-access::rect o (y0 y1)
                    (set! y1 (+ y0 v)))))))
```

Setting the `width` virtual slot (resp. the `height` slot) automatically adjusts the `x1` value (resp. the `y1` value) and *vice versa*. No memory is allocated for `width` and `height`, as their *values* are computed each time they are accessed.

3 The Biglook library

Biglook is an Object Oriented Scheme library for constructing GUIs. It offers an extensive set of widgets such as labels, buttons, text editors, gauges, canvases. Most of the functionality it offers are available through classes rather than through *ad-hoc* functions. For instance, instead of having the classical functions `iconify`, `deiconify` and `window-iconified?` for the `window` widget, Biglook offers the virtual slot `visible` to implement this functionality. Setting the `visible` slot enables iconification/deiconification. Reading the `visible` slot unveils the window iconification state. As we will see, this design choice yields a simpler API and allows usage of introspective techniques for GUIs programming.

In Section 3.1 we first present a small interface and discuss how to create the widgets which compose it in Section 3.2. Section 3.4 describes the notion of *container* widget and placement rules. Finally, in Section 3.5 we show how to make a widget reactive to an external event such as a mouse click.

Throughout this section we justify the choices we have made when designing the Biglook API.

Figure 2: A simple example

Our reflection on how to create a widget or how to handle interfaces events are presented in Sections 3.3 and 3.6.

3.1 A Biglook example

Biglook uses a declarative model for constructing GUIs. This permits a clear separation between the code of the interface and the code of the application. The construction of an interface starts by declaring the various widgets which compose it. Then, the behavior of each widget is specified independently of its creation by associating an action (a Scheme closure) with a widget specific event (key pressed, mouse click, mouse motion, ...).

```
(module example (library biglook))

(define awin
   (instantiate::window
      (title "A Biglook example")))
(define acheck
   (instantiate::check-button
      (parent awin)
      (text "underline")))
(define aradio
   (instantiate::radio
      (parent awin)
      (orientation 'horizontal)
      (border-width 2)
      (texts '("bold" "italic" "plain"))))
(define abutton
   (instantiate::button
      (parent awin)
      (relief 'flat)
      (text "Quit")))
```

Figure 3: A simple example, the source code

Figure 2 is a screen shot of a simple Biglook application. It is made of a window (here named “A Biglook example”), a check button (the toggle button underline), a radio button group (bold, italic, plain), and a plain button (Quit). The source code of this example is given Figure 3. From that code, we see that in order to access Biglook classes and functions, the program uses a library module clause line *1*. This program creates a window (line *4*), and three widgets (line *7*, line *11* & *17*).

In the sections 3.2 and 3.4 we present how to create widgets and how to place them in a window. The Figure 2 interface is inert, that is, no action is associated with the widgets yet. We will see in Section 3.5 how actions can be associated with widgets.

3.2 Widget Creation

The graphical objects (i.e., *widgets*) defined by the Biglook library such as menus, labels or buttons are represented by Bigloo classes. Each class defines a set of slots that implement the configuration of the instances. Consequently, tuning the look of a widget consists in assigning correct values to its slots. The library offers *standard* default values for each widget but these values can of course be changed. Generally the customization is done at widget creation time. For instance, the radio group of Figure 3 will be displayed horizontally and with a border size of 2 pixels (lines *13* and *14*). A particular aspect of a widget can be changed by setting a new value to its corresponding slot. For instance, the expression

```
(radio-orientation-set! aradio 'vertical)
```

changes the orientation of the radio group forcing a re-display of the whole window. Of course, the value of this slot can be retrieved by just reading it:

```
(print (radio-orientation aradio))
```

Remember that, as seen in Section 1.2.2, instead of using the functions created by Bigloo that fetch and write the value of a slot, one may write an equivalent program using the Bigloo special form `with-access`:

```
(with-access::radio aradio (orientation)
   (set! orientation 'vertical)
   (print orientation))
```

In the rest of this paper, we will use either forms for accessing class slots.

3.3 Reflection on widget creation

Biglook widget creation supports variable number of arguments and keyword parameters (parameters that can be passed in any order because their actual value is associated with a name). We have found these features very useful in order to enable declarative programming for GUI applications. For instance, a plain Biglook button is characterized by 20 slots. Some of them describe the graphical representation (colors, border sizes, . . .), some others describe the internal state of the button (widget parent, associated value, associated text, . . .). In general, these numerous slots have default values. When an instance is created, only slots that have no default values must be provided. Slots are initialized with their default value unless a user value is specified. As a consequence, the form that operates widget construction (*e.g.*, class instantiation) must accept a variable number of arguments. Only some slot values must be provided, others are optional. In addition, because widget constructors accept a large number of parameters, it is convenient to *name* them and to be able to pass them in any order. This is made possible in Biglook by the `instantiate::` form. As this form is implemented using macros that are expanded into calls to the class constructors where each declared slot is provided with a value, there is no run-time overhead associated with forms such as `instantiate::`.

Lacking variable number of arguments or keywords disables declarative programming style for GUIs because widgets have to be created and, in a second step, specific attributes have to be provided. Even overloading and class constructors do not help. Let's suppose our window classes implemented in Java AWT [15] or Swing [29]. To enable a full declarative style, we should provide the button class definition with 2^{20} constructors (a constructor for each possible combination of provided slots). Even for much smaller classes, this is impractical because, in general, overloading dispatches on types only and several slots can have the same type. For instance, imagine that we want to change the graphical appearance of our window. Instead of using the smallest area large enough to display the three widgets, we want to force the width of the window to a specific value. We can turn the definition of Figure 3, line *4* to:

```
(define awin
   (instantiate::window
      (title "A Biglook example")
      (width 300)))
```

If we want to specify both width and height, we can use:

```
(define awin
   (instantiate::window
      (title "A Biglook example")
      (width 300)
      (height 200)))
```

Languages relying on type overloading cannot propose these different constructors because the width and the height of a window are of the same type.

Languages without overloading nor n-ary functions traditionally use lists to collect optional and keyworded arguments. In addition to the runtime cost imposed by the list constructions, the called function has to dispatch, at runtime, over the list to set the parameters values. Furthermore, such a call cannot be statically typed anymore.

3.4 Containers and widget placement

Biglook uses special sort of widgets to enable user customized widget placements: the `container` class. A container is a widget that can embed other widgets. Those widgets are called the *children* of the container. For instance, a window such as the one defined line *4* of Figure 3, is a container, it "contains" the three other widgets. With the exception of windows, all widgets must be associated with a container in order to be visible on the screen. To associate a widget with a container, one have to set its `parent` slot (see lines *8*, *12* & *18* of Figure 3). Biglook proposes several kind of containers: aligned containers (such as boxes and windows), grid containers, note pad containers, paned containers, containers with scrollbars, etc.

Let us present here two examples of containers. First, let us consider that we want to modify the interface of Figure 2. We want the buttons to be displayed horizontally instead of vertically

Figure 4: An horizontal layout

(see Figure 4). For that, we add a new container in the window, an horizontal `box`:

```
(define awin
   (instantiate::window
      (title "A Biglook example")))

(define abox
   (instantiate::box
      (parent awin)
      (orientation 'horizontal)))

(define acheck
   (instantiate::check-button
      (parent abox)
      (text "underline")))
...
```

For the second example, we combine containers to design complex interfaces. For instance, the interface of Figure 5 can be implemented as:

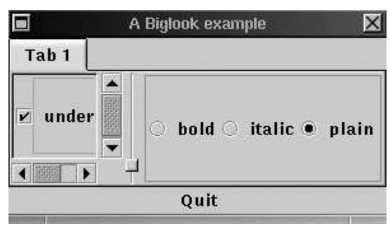

Figure 5: Several containers

```
(define awin
   (instantiate::window
      (title "A Biglook example")))

(define atab
   (instantiate::notepad
      (parent awin)))

(define apane
   (instantiate::paned
      (orientation 'horizontal)
      (parent atab)))

(define ascroll
   (instantiate::scroll
      (parent apane)))
```

```
17:(define acheck
18:   (instantiate::check-button
19:      (parent ascroll)
20:      (text "underline")))
21:
22:(define aradio
23:   (instantiate::radio
24:      (parent apane)
25:      (border-width 2)
26:      (orientation 'horizontal)
27:      (texts '("bold" "italic" "plain"))))
28:
29:(define abutton
30:   (instantiate::button
31:      (parent awin)
32:      (relief 'flat)
33:      (text "Quit")))
```

Note that even if the interfaces of Figures 2 and 5 seem quite different, we only need to modify the *parent* slot of the `acheck` and `aradio` widgets to embed them in the new containers `atab` (line *6*), `apane` (line *10*) and `ascroll` (line *15*).

3.5 Event Management

Biglook widgets allow the creation of complex GUIs with minimal efforts. In general, when building such interfaces, one of the main difficulties lies in trying to separate the code of the interface from the rest of the program. Making the GUI code independent from the rest of the application is important because:

- GUIs are often built on a trial-fail basis. It is hard to conceive an interface ex-nihilo, and it is generally after using it for a while that the elements of the GUI find their place. Keeping the code independent from the rest of the application allows the development of prototypes of the interface without nasty consequences on the other parts of the program.
- A given program can have several interfaces according to the device on which it is run (e.g. graphical screen, PDA, alphanumeric terminal). With an independent interface code, different interfaces can be connected to the same program.
- The GUI of an application can be constructed interactively by an interface builder. In such a case, it is preferable to keep the mechanically generated code separate from hand written parts of the application.

Graphical events (mouse click, key pressed, window destruction ...) can be associated with widgets by the means of the widgets `event` slot. This slot must contain an instance of the class `event-handler` which is defined as:

```
(class event-handler
   (configure::procedure (default ...))
   ;; window events
   (destroy::procedure (default ...))
   ...
   ;; mouse events
   (press::procedure (default ...))
   (enter::procedure (default ...))
   ...
   ;; keyboard events
   (key::procedure (default ...))
   ...)
```

Each slot of an event-handler is a procedure called a *call-back* that accepts one argument. When a graphical action occurs on a widget, the associated call-back is invoked passing it an *event descriptor*. Those descriptors are allocated by the Biglook runtime system. They are instances of the `event` class which is defined as:

```
(class event
   ;; the widget which receives the event
   (widget::widget read-only)
   ;; the button number or -1
   (button::int read-only)
   ;; the modifiers list (e.g. shift)
   (modifiers::pair-nil read-only)
   ;; the x position of the mouse
   (x::int read-only)
   ;; the y position of the mouse
   (y::int read-only)
   ;; the character pressed or -1
   (char::char read-only)
   ...)
```

So, modifying the example of Figure 3 for the Quit button to be aware of mouse button 1 clicks, we could write the code as follows:

```
1:(let* ((p (lambda (e)
2:            (if (= (event-button e) 1)
3:   (exit))))
4:        (evt (instantiate::event-handler
5:                (press p))))
6:   (with-access::button abutton (event)
7:      (set! event evt)))
```

That is, on line *4* we allocate `evt`, an instance of the `eventhandler` class. That event handler is connected to the button line *7*. The event handler only reacts to mouse `press` events. When such an event is raised, the call-back line *1* is invoked, its formal parameter e being bound to an instance of the `event` class. This function checks the button number of the raised event (line *2*). When the first button is pressed the Biglook application exits.

It is possible to modify already connected call-backs. For instance, if we want the Quit button to emit a sound when it is pressed. we can write:

```
(with-access::button abutton (event)
   (with-access::event-handler event (press)
      (let ((olde press))
  (set! press
        (lambda (e)
   (beep) (olde e))))))
```

Since widgets call-backs are plain Scheme closures, they can be manipulated as first class objects, as in this example where new call-backs capture the values of the old call-backs in order to reuse them.

3.6 Reflection on event handling

Most modern widget toolkits (with the exception of Qt [5]) use a call-back framework. That is, user commands are associated with specific events (such as mouse click, mouse motion, keyboard inputs, ...). When an event is raised, the user command is invoked. We think that the closure mechanism is the most simple and efficient way to implement call-backs even if some alternatives exist.

The rest of this section discusses how call-backs can be implemented depending on the features provided by the host language used to implement a graphical toolkit.

3.6.1 Languages that support functions without environment

ISO-C [16] supports global functions but no local functions. A C function is always top-level and may only access its parameters and the set of global variables. C functions have no definition environment. However, without an environment, a call-back is extremely restricted. In particular, a call-back is likely to access the widget that owns it. In GTK+ (a C toolkit) when a call-back is associated with an event, an optional value may be specified that will be passed to the call-back when the event is raised. This user value actually *is* the environment of the call-back. GTK+ mimics closures with its explicit parameter-passing scheme. We may notice that the allocation and the management of the closure environment is in charge of the client application.

3.6.2 Languages with classes

For languages with classes such as Java, another strategy can be used. Call-backs may be implemented using class member functions. Member functions may access the object and the object's attributes for which they are invoked. Member functions look like closures. However, member functions are not closures because they are associated with classes. In other words, all the instances of a class share the implementation of all their member functions. That is, different call-back implementations require different class declarations. For instance, if one wants to implement a button with a call-back printing a plain message and another one emitting a sound, two classes have to be defined. These class declarations turn out to be an hindrance to simplicity and readability. In addition, if several events must be handled by one widget, this technique turns out to be impractical because it is not possible to define a new class for each kind of events the widget must react to (mouse-1, mouse-2, mouse-3, shift-mouse-1, ctrl-mouse-1, shift-ctrl-mouse-1, ...).

To avoid these extra class definitions, Java has introduced inner classes. An inner class is a class defined inside another class; it may be anonymous. Because in GUI programming inner classes are used to implement call-backs and as they are numerous, Java proposes a new syntax that enables within a single expression, to declare and to instantiate an inner classe. For instance:

```
button.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent e) {
      a user code that may reference
      current lexical bindings
   }
})
```

The expression `new ActionListener...` deserves some explanation: *1)* it declares an anonymous inner class that *2)* implements the interface `ActionListener` and *3)* it creates one unique instance of that new class that is sent to the method `addActionListener` of the `Button` instance. That is, anonymous inner classes are the exact Java implementation of closures. It is worth pointing out that Scheme syntax for closures is far more compact than the Java one.

3.6.3 Closures

Closures are central to GUI programming because they are one of the most natural way to implement call-backs. As we have seen, all call-back based toolkits offer a mechanism similar to closures. It can be member functions or anonymous Java classes or extra parameter passed to C functions. However, we have found that solutions of non-functional programming languages are not as convenient as Scheme's `lambda` expression because either the user is responsible of the construction of the object representing the closure or extra syntax is introduced.

```
1:(define tree1
2:   (instantiate::tree
3:      (parent apane)
4:      (root object)
5:      (node-label class-name)
6:      (node-children class-subclasses)))
7:(define tree2
8:   (instantiate::tree
9:      (parent apane)
10:     (root "/")
11:     (node-label (lambda (x) x))
12:     (node-children directory->list)))
```

Figure 6: Two Biglook trees

Not only call-backs are easily implemented by the means of closures but we have also found that closures are handy to implement pre-existing data structure visualization. Consider the screen shot, Figure 7. It is made of two Biglook trees. A Biglook tree is a visualization of an existing data structure. That is, a Biglook tree does not contain data by itself. It only *visualizes* an existing structure. A Biglook tree is defined by three slots: *i)* the root of the tree (`root`), *ii)* a function that extracts a string which stands for the label of a node (`node-label`), *iii)* a function that computes the children list of a node (`node-children`). The declaration of the trees of Figure 7 are given Fig-

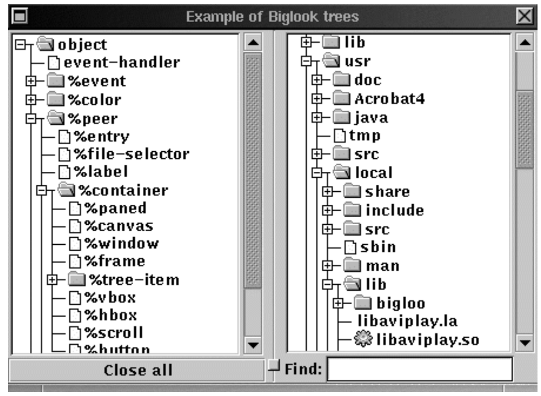

Figure 7: Two Biglook trees

ure 6, lines *2* and *8*. The first tree (`tree1`) is a class tree, its root is the Scheme `object` class denoting the root of the inheritance tree (line *4*). The second tree (`tree2`) is a file tree. Its root is `"/"`, the root of the file system (line *10*). To compute the name of a node representing a class it is only required to extract the name of that class (line *5*). The name of a node representing a directory is the node itself since the node is a string (line *11*). The children list of a class is computed using the Bigloo library function `class-subclasses` (line *6*). The children list of a directory is computed by the library function `directory->list` (line *12*). As one may notice "hooking" a tree to a data structure is a simple task. The resulting program is compact. We think that this compactness is another strength of Scheme closures.

In general, the Biglook API enables small source codes for GUIs. On a typical graphical interface we have found that the Biglook source is about twice smaller than the same interface implemented in Tcl/Tk and about four times smaller than the interface implemented in C with GTK+.

4 Implementing Biglook

In this section, we present the overall Biglook architecture and implementation. Then, we detail the role of *virtual slots*.

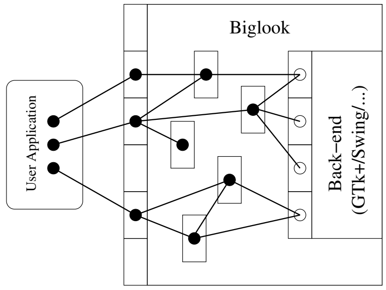

Figure 8: The Biglook software architecture

4.1 Library Architecture

The Biglook library is implemented on top of a native *back-end* toolkit (currently a GTK+ back-end and a Swing back-end are available). It takes advantage of the efficiency of the *back-end* low level operations and it imposes a low memory overhead (about 15% of additional allocations). The difference in execution time is marginal. As a consequence, Biglook can be used to implement applications using complex graphical interfaces such as the one presented Figure 1. Interested readers can find precise performance evaluation of these implementations in a previous technical report [12].

Because Biglook makes few assumptions about the underlying toolkit, re-targeting its implementation to another back-end is generally possible. To be a potential Biglook back-end, a toolkit must provide: *i)* a way to identify a particular component of the interface. Of course all the toolkits provide this, even if the representation used can differ, such as an integer, a string, a function, and so on. *ii)* an event manager that does not hide any events. All the toolkits we have tried satisfy this criteria (Tk, GTK+ and Swing). In addition, if a back-end lacks some widgets, Biglook implements them using primitive widgets. Figure 8 shows the underlying architecture of the Biglook toolkit.

Actual graphical toolkits generally support these prerequisites and, as such, can be used as potential Biglook back-end. We will see in Section 4.2 that using virtual slots allows the building of the rest of the library on this minimal basis.

4.2 Virtual Slots and Biglook

A simple widget is a widget that is directly mapped into a builtin widget. All simple widgets are implemented according to the same framework: they inherit from the `widget` class and they define user customization options. These options are implemented using virtual slots. We present here a possible implementation of the `label` class. For the sake of simplicity, we assume that this class extends the `widget` class with only one additional slot: the `text` slot that specifies the label text.

```
(class widget::object
   (builtin-widget read-only)
   ...)

(class label::widget
   (text
    (get
     (lambda (o::label)
 (with-access::label o (builtin)
    (<builtin-label-text> builtin))))
    (set
     (lambda (o::label v::procedure)
 (with-access::label o (builtin)
    (<builtin-label-text-set!> builtin
 v))))))
```

Implementing the `text` slot requires virtual slots. Its getter and setter functions directly interact with the back-end toolkit. Virtual slots are used to establish the connection between Biglook user point of view of widgets and their native implementation. Virtual slots are used to provide an object oriented class-based API to Biglook, independent of the back-end.

Now that the slots of a label widget are defined, we must define how such a widget has to be initialized. The class initialization specific code is given to the system via the generic function `realize-widget`. For each class of the library a method overrides this generic function and must call the back-end to create the graphical object associated with the class. For a label, the method we need to write is:

```
(define-method (realize-widget o::label)
   (with-access::label o (builtin parent)
      (set! builtin
            (<builtin-make-label> parent))))
```

This method creates a builtin label via the low level *<builtin-make-label>* function and stores the result in the `builtin` slot. This slot creates the link between the Biglook toolkit and the back-end.

5 Related work

Many functional languages are connected to widget libraries especially to the Tk toolkit. Few of them use object-oriented programming except in the Scheme world, we can cite mainly STk, SWL and MrEd.

5.1 Scheme widget libraries

STk [11] is a Scheme interpreter extended with the Tk graphical toolkit. To some extent, STk is the ancestor of Biglook, since it was developed by one of the authors of this paper. However, STk is tightly coupled to the Tk toolkit and even if this toolkit is presented to the user through an object oriented API as in Biglook, no provision was made to be independent from this back-end.

SWL is a contribution to the Petite Chez Scheme system [6]. It relies on an interpreter and uses Tk as back-end. In SWL, native Tk widgets are mapped to Chez Scheme classes and in this respect this toolkit is similar to the Biglook or STk libraries. However, SWL implementation is very different from the one used by those libraries since SWL widgets explicitly *talk* with a regular Tcl/Tk evaluator.

MrEd [9], a part of the DrScheme project [7], is a programming environment that contains an interpreter, a compiler and other various programming tools such as browser, debugger, etc. The back-end toolkit used by MrEd is wxWindow [27], a toolkit that is available under various platforms (Unix, Windows, etc.). As STk, this toolkit is completely dependent of its back-end.

5.2 Other functional languages widget libraries

Other functional languages provide graphical primitives using regular functions. The main contribution for strict functional programming languages has been developed for the Caml programming language [30].

The first attempt, CamlTk, is quickly surveyed in [23]. The design of CamlTk is different from the one of Biglook. CamlTk *binds* Tk functions in Caml while Biglook provides an original API made of classes.

Recently a new widget library based on GTK+ has been proposed for Caml. No article describes that connection. However, some work has been described to add keyword parameters to Caml in order to help the connection with widget libraries [10]. The philosophy of that work differs from ours because that new library makes the GTK+ API available from Caml. No attempts are made to present a neutral API as we did for Biglook.

Programming graphical user interfaces with lazy languages is far more challenging than with strict functional languages. The problem is to tame the imperative aspects of graphical I/O in such languages [18, 28]. Several solutions have been proposed: Fudget by Carlsson and Hallgren [2, 3], Haggis by Finne and Peyton Jones [8], TkGofer by Vullings and Claessen [4] and, more recently the extension of the former TclHaskell library: FranTk by Sage [24].

Conclusion

In this paper we have presented Biglook, a widget library for the Bigloo system. The architecture of Biglook enables different ports. Currently two ports are available: a GTK+ port and a Swing port. Biglook source code can be indifferently linked against the two libraries. Biglook API uses an object oriented programming style to handle graphical objects and a functional oriented programming style to implement the interface reactivity. We have found that combining the two programming styles enables more compact implementations for GUIs than most of the other graphical toolkits.

Acknowledgments

Many thanks to Keith Packard, Jacques Garrigue, Didier Remy, Peter Sander, Matthias Felleisen, Simon Peyton Jones and to Céline for their helpful feedbacks on this work.

References

[1] D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. **Common lisp object system specification**. In *special issue*, number 23 in SIGPLAN Notices, September 1988.

[2] M. Carlsson and T. Hallgren. FUDGETS **- A Graphical User Interface in a Lazy Functional Language**. In *FPCA '93 - Conference on Functional Programming Languages and Computer Architecture*, pages 321–330. ACM Press, June 1993.

[3] M. Carlsson and T. Hallgren. **Fudgets — Purely Functional Processes with applications to Graphical User Interfaces**. PhD thesis, Department of Computing Science, Chalmers University of Technology, S-412 96 Gteborg, Sweden, March 1998.

[4] K. Claessen, T. Vullinghs, and E. Meijer. **Structuring Graphical Paradigm in TkGofer**. In *Int'l Conf. on Functional Programming*, 1997.

[5] M. Dalheimer. **Programming with Qt**. O'Reilly, 1st edition, april 1999.

[6] K. Dybvig. **Chez Scheme User's Guide**. Cadence Research Systems, 1998.

[7] M. Felleisen, R. Findler, M. Flatt, and S. Krishnamurthi. **The DrScheme Project: An Overview**. SIGPLAN *Notices*, 1998.

[8] S. Finne and S. Peyton Jones. **Composing Haggis**. In *Fifth Eurographics Workshop on Programming Paradigm for Computer Graphics*, Maastricht, NL, September 1995.

[9] M. Flatt, R. Findler, S. Krishnamuryhi, and M. Felleisen. **Programming Languages as Operating Systems (*or* Revenge of the Son of the Lisp Machine)**. In *Int'l Conf. on Functional Programming*, Paris, France, 1999.

[10] J. Furuse and J. Garrigue. **A label-selective lambda-calculus with optional arguments and its compilation method**. Technical Report RIMS Preprint 1041, Research Institute for Mathematical Sciences, Kyoto University, October 1995.

[11] E. Gallesio. **STk Reference Manual**. Technical Report RT 95-31a, I3S-CNRS/Univ. of Nice–Sophia Antipolis, July 1995.

[12] E. Gallesio and M. Serrano. **Graphical user interfaces with Biglook**. Technical Report I3S/RR–2001-13–FR, I3S-CNRS/Univ. of Nice–Sophia Antipolis, September 2001.

[13] A. Goldberg and D. Robson. **Smalltalk-80: The Language and Its Implementation**. Addison-Wesley, 1983.

[14] J. Gosling, B. Joy, and G. Steele. **The Java™ Language Specification**. Addison-Wesley, 1996.

[15] J. Gosling, F. Yellin, and the Java Team. **The Java^{TM} Application Programming Interface, Volume 2: Window Toolkit and Applets**. Addison-Wesley, 1996.

[16] ISO/IEC. **9899 Programming Language - C**. Technical Report DIS 9899, ISO, July 1990.

[17] R. Kelsey, W. Clinger, and J. Rees. **The Revised(5) Report on the Algorithmic Language Scheme**. *Higher-Order and Symbolic Computation*, 11(1), September 1998.

[18] R. Noble and C. Runciman. **Functional Languages and Graphical User Interfaces – a review and a case study**. Technical Report 94-223, Department of computer Science, University of York, February 1994.

[19] J. Ousterhout. **Tcl and the Tk toolkit**. Addison-Wesley, 1994.

[20] H. Pennington. **Gtk+/Gnome Application Development**. New Riders Publishing, 1999.

[21] C. Queinnec. **Designing** MEROON **V3**. In *Workshop on Object-Oriented Programming in Lisp*, 1993.

[22] D. Rémy and J. Vouillon. **Obective ML: A simple object-oriented extension of ML**. In *Symposium on Principles of Programming Languages*, pages 40–53, 1997.

[23] F. Rouaix. **A Web navigator with applets in Caml**. In *Proceedings of the 5th International World Wide Web Conference, in Computer Networks and Telecommunications Networking*, volume 28:7–11, pages 1365–1371. Elsevier, May 1996.

[24] M. Sage. **FranTk – A declarative GUI language for Haskell**. In *Int'l Conf. on Functional Programming*, Montral, Qubec, Canada, September 2000.

[25] M. Serrano. **Bigloo user's manual**. RT 0169, INRIA-Rocquencourt, France, December 1994.

[26] M. Serrano. **Wide classes**. In *Proceedings ECOOP'99*, pages 391–415, Lisbon, Portugal, June 1999.

[27] J. Smart. **wxWindows toolkit Reference Manual**. available at `http://web.ukonline.co.uk/julian-.smart/wxwin`, 1992.

[28] T. Vullings, D. Tuijnman, and V. Schulte. **Lightweight GUIs for Functional Programming**. In *Int. Symp. on Programming Languages, Implementations, Logics, and Programs*, 1995.

[29] K. Walrath and M. Campione. **The JFC Swing Tutorial: A Guide to Constructing GUIS**. Addison-Wesley, July 1999.

[30] P. Weis and *al.* **The CAML Reference manual**. Technical Report 121, INRIA-Rocquencourt, 1991.