Programming Graphical User Interfaces with Scheme

Erick Gallesio and Manuel Serrano
{Erick.Gallesio,Manuel.Serrano}@unice.fr
http://kaolin.unice.fr/~{gallesio,serrano}

Université de Nice Sophia-Antipolis
650, route des Colles, B.P. 145
F-06903 Sophia-Antipolis, CEDEX

Abstract

This paper presents Biglook, a widget library for an extended version of the Scheme programming language. It uses classes of a CLOS-like object layer to represent widgets and Scheme closures to handle graphical 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.

Introduction

Functional languages (henceforth FLs) must provide facilities for constructing contemporary applications. For instance, FLs *must* provide facilities implementing database access, network roaming or graphical user interfaces (henceforth GUI) (Gabriel, 1991; Wadler, 1998). We have studied the problem of constructing GUI in functional languages by designing a widget library for the Scheme programming language, called Biglook, which is presented in this paper. Biglook's primary use is to implement graphical applications (applications such as `xload`, editors *à la* Emacs, browser *à la* Netscape or even 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.

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 (Serrano, 1994). 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+ (Pennington, 1999) associated with the Bigloo C back-end and Swing (Walrath & Campione, 1999) with the Bigloo

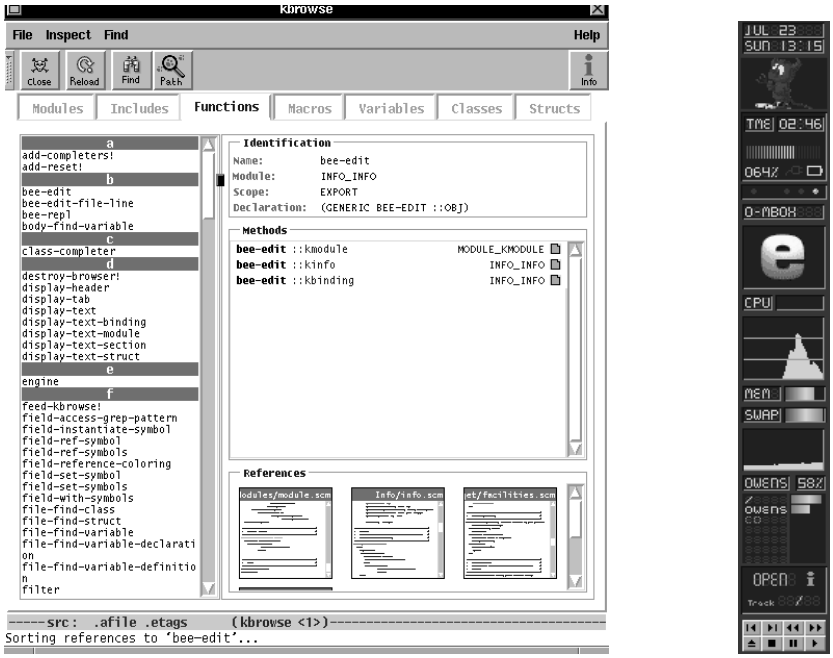


Fig. 1. Two Biglook applications: a code browser on the left, "dock" applications on the right

JVM back-end. The previous release of Biglook (Gallesio & Serrano, 2001) used Tk (Ousterhout, 1994).

Bigloo implements an object layer inspired by CLOS (Bobrow *et al.*, 1988). It is a class-based model where methods override generic functions and are not declared inside classes as in Smalltalk (Goldberg & Robson, 1983), O'Caml (Rémy & Vouillon, 1997) or Java (Gosling *et al.*, 1996b).

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 (such as 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 by looking at a simple Biglook application and its associated source code. Then, we detail the Biglook programming princi-

ples (widgets creation, event handling, etc.) motivating our design orientations by programming language considerations. In this section we compare the functional programming style and the object oriented one. In Section 4 we present the Biglook implementation. Finally, 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 implementation defined by (Kelsey *et al.*, 1998). 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 functions: 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+)*
)
optional-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 a merely a collection of pre-compiled modules. Using a library is equivalent to *import-*

ing all the modules composing the library. Detailed information on Bigloo modules may be found in (Serrano, 1994; Serrano, 1999).

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 (Bobrow *et al.*, 1988) inspired, to a great extent, by MEROON (Queinnec, 1993).

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 optional-init optional-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 (*optional-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`*), may be immutable (if declared `read-only`), 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 ...) optional-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
```

Generic functions is a well known programming paradigm introduced by the Common Lisp object system. Generic functions provide a greater flexibility and extensibility than method-in-class systems, enabling users to customize libraries in many directions. Incarnations of generic functions exist in several programming language (Dylan (Shalit, 1996), Cecil multi-methods (Chambers, 1993), O'Haskell classes and instances (Nordlander, 1999), ...).

2 Virtual slots

Bigloo supports two kind of instance slots: regular slots, that have 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 `rectangle` 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 rectangle
   x0 y0 x1 y1
   (width (get (lambda (o)
                  (with-access::rectangle o (x0 x1)
                     (- x1 x0))))
          (set (lambda (o v)
                  (with-access::rectangle o (x0 x1)
                     (set! x1 (+ x0 v))))))
   (height (get (lambda (o)
                   (with-access::rectangle o (y0 y1)
                      (- y1 y0))))
           (set (lambda (o v)
                   (with-access::rectangle o (y0 y1)
                      (set! y1 (+ y0 v)))))))
```

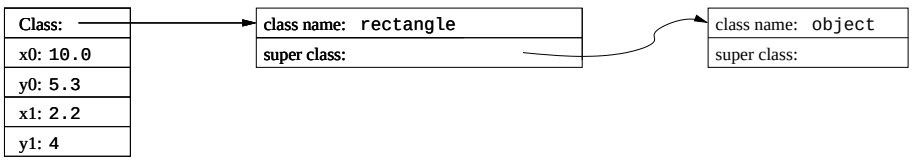
Setting the `width` virtual slot (resp. the `height` slot) automatically adjusts the `x1` value (resp. the `y1` value) and *vice versa*. No memory is allocated for `width` and `height`, as their *values* are computed each time they are accessed.

The major strength of virtual slots is that arbitrary computations can take when accessing or setting such a slot. For instance, one may imagine that accessing a virtual slot actually fetches and stores a data from a database or access a remote data trough the network.

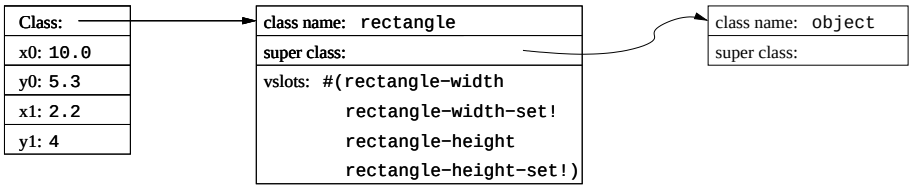
2.1 Virtual Slots Implementation

In a language provided with a Meta Object Protocol (henceforth MOP) (Kiczales *et al.*, 1992; Gallesio, 1996), virtual slots implementation is straightforward. Meta object programming makes compilation harder and therefore Bigloo does not include it.

Declaring a standard class defines the memory layout of its instances. Each class slot is allocated a memory area. For instance, if we consider a rectangle class *without* `width` and `height` virtual slots, the memory layout of the instances could be:



Each class declaring virtual slots contains a vector of virtual slots accessors. So, the memory layout for instances of the `rectangle`, *provided with* the `width` and `height` virtual slots is:



Accessing virtual slots fetches from the class virtual vector the correct function (the offset is computed statically) and calls it. Consequently, accessing a virtual slot of an instance `i` is:

```
(let ((vvec (class-virtual-vector (object-class i))))
   ((vector-ref vvec <a-static-offset>) i))
```

Obviously, accessing virtual slots is more expensive than accessing regular ones: the overhead is one additional memory access and one additional function call. However, the extra memory read can be removed if each instance is provided with an extra slot pointing to the virtual vector of its class. Bigloo does not use this framework to maintain class instances as small as possible.

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. Thus, it is our opinion that 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. In Section 3.5 we show how to make a widget reactive to an external event such as a mouse click. Finally, in Section 3.7 we show how a user can extend the library and create a complex widget by composing simpler ones.

Throughout this section we justify the choices we have made when designing the Biglook API. Our reflection on how to create a widget or how to handle interfaces events are presented in Sections 3.3 and 3.6. In Section 3.8, we discuss more generally the problem of widget composition and the role of generic functions in this area as well as in library extension.

3.1 A Biglook example

Biglook uses a declarative model for the construction of 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, ...).

Fig. 2: A simple example

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.

```
1:  (module example (library biglook))
2:
3:  (define awin (instantiate::window
4:                  (title "A Biglook example")))
5:  (define acheck (instantiate::check-button
6:                    (parent awin)
7:                    (text "underline")))
8:  (define aradio (instantiate::radio
9:                    (parent awin)
10:                   (orientation 'horizontal)
11:                   (border-width 2)
12:                   (texts '("bold" "italic" "plain"))))
13: (define abutton (instantiate::button
14:                    (parent awin)
15:                    (relief 'flat)
16:                    (text "Quit")))
```

Fig. 3. A Simple example, the source code

From that code, we see that to access Biglook classes and functions, the program uses a library module clause line *1*. This program creates a window (line *3*), and three widgets (line *5*, line *8* & *13*).

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 *10* and *11*). 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 (Gosling *et al.*, 1996a) or Swing (Walrath & Campione, 1999). 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 *3* 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 list 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 Figure 3, line *3* is a container, it "contains" the three other widgets. Exception made 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 *6*, *9* & *14* 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 (see Figure 4). For that, we add a new container in the window, an horizontal `box`:

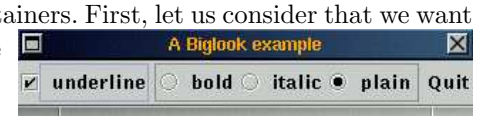


Fig. 4: An horizontal layout

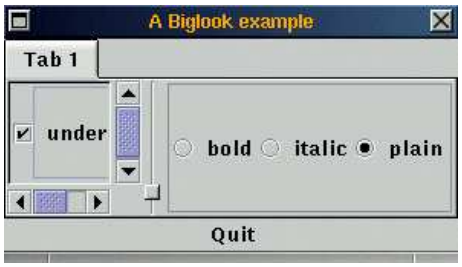


Fig. 5. Several containers

```
1:  (define awin (instantiate::window
2:                  (title "A Biglook example")))
3:  (define abox (instantiate::box
4:                  (parent awin)
5:                  (orientation 'horizontal)))
6:  (define acheck (instantiate::check-button
7:                    (parent abox)
8:                    (text "underline")))
9:  ...
```

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

```
1:  (define awin (instantiate::window
2:                  (title "A Biglook example")))
3:  (define atab (instantiate::notepad
4:                  (parent awin)))
5:  (define apane (instantiate::paned
6:                   (orientation 'horizontal)
7:                   (parent atab)))
8:  (define ascroll (instantiate::scroll
9:                    (parent apane)))
10: (define acheck (instantiate::check-button
11:                   (parent ascroll)
12:                   (text "underline")))
13: (define aradio (instantiate::radio
14:                   (parent apane)
15:                   (border-width 2)
16:                   (orientation 'horizontal)
17:                   (texts '("bold" "italic" "plain"))))
18: (define abutton (instantiate::button
19:                    (parent awin)
20:                    (relief 'flat)
21:                    (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 *3*), `apane` (line *5*) and `ascroll` (line *8*).

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 dummy-callback))
   ;; window events
   (destroy::procedure (default dummy-callback))
   ...
   ;; mouse events
   (press::procedure (default dummy-callback))
   (enter::procedure (default dummy-callback))
   ...
   ;; keyboard events
   (key::procedure (default dummy-callback))
   ...)
```

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 shown in Figure 6.

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

Fig. 6. An event handler

That is, on line *1* we allocate `evt`, an instance of the `event-handler` class. That event handler is connected to the button line *5*. The event handler only reacts to mouse `press` events. When such an event is raised, the call-back line *2* 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 *3*). When the first button is pressed the Biglook application exits.

Since it is frequent that GUI widgets react to mouse button 1 clicks, Biglook widgets propose a shorthand to associate a call-back with that particular event: widgets are provided with a `command` slot. Setting that slot is equivalent to the code sequence shown in Figure 6. On the other hand, Biglook event handlers are handy when: *i)* several widgets have to be connected to the same call-backs (such as a minesweeper game where all the buttons share the very same behaviors) or *ii)* several call-backs have to be connected to a widget. For instance, suppose that we want our Quit button to adopt a different appearance when the mouse cursor is flying over that button, we simply write:

```
(with-access::button abutton (event relief)
   (with-access::event-handler event (enter leave)
      (let ((oldr relief))
        (set! enter (lambda (e) (set! relief 'shadow-in)))
        (set! leave (lambda (e) (set! relief oldr))))))
```

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

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

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 ones in order to reuse them.

3.6 Reflection on event handling

Most modern widget toolkits (exception made of Qt (Dalheimer, 1999)) 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 (ISO/IEC, 1990) 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.

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 8. It is made of two Biglook

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

Fig. 7. Two Biglook trees

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 8 are given Figure 7, lines *1* and *6*. The first tree (`tree1`) is a class tree, its root is the Scheme `object` class denoting the root of the inheritance tree (line *3*). The second tree (`tree2`) is a file tree. Its root is `"/"`, the root of the file system (line *8*). To compute the name of a node representing a class it is only required to extract the name of that class (line *4*). The name of a node representing a directory is the node itself since the node is a string (line *9*). The children list of a class is computed using the Bigloo library function `class-subclasses` (line *5*). The children list of a directory is computed by the library function `directory->list` (line *10*). 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.

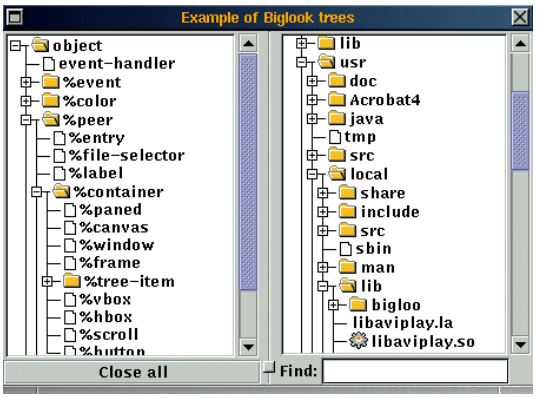


Fig. 8: Two Biglook trees

3.7 Widgets composition

Biglook clients can define their own sets of widgets. New widgets can be made by composing and graphically tuning simpler ones. A composite widget is completely written in Scheme and is generally built by inheriting from an existing widget. One of the simplest composite widget is a *labeled entry*. Such a widget is a small line editor with a label on its left indicating the kind of value that must be entered in the editor.

These widgets can be easily implemented with two basic builtin widgets: a *label* and an *entry*. The `labeled-entry` class is defined in the Biglook library, and we show here how to implement a simplified version.

The main difficulty is to determine the inheritance scheme to be used. Often, composite widgets are standard widgets augmented with additional graphical material (for instance a *text with a scrollbar* widget inherits its behavior from `text` and not from `scrollbar`). For the `labeled-entry` we consider it as a specialized version of an `entry` with a `label` as decoration. This observation yields to the structure of the `labeled-entry` class: it inherits from the `entry` class and provides a slot containing a reference to a `label`. The last design issue is to select which slots

are defined for `labeled-entry` instances. They can be characterized according to three categories:

- The ones that change the properties of the inherited class. This is a classical object problem that can be solved by a slot redefinition. In the case of a *labeled entry*, for instance, we could wish to redefine the `relief` property (which is a graphical attribute of widgets) to its container rather than the one of the entry, as implied by our inheritance scheme.
- The ones that give access to a property of a component of the composite which is not the main one (i.e. a decoration element). This can be easily achieved with a virtual slot that accesses a property of a real slot of the widget. For instance, the slot `title` of a labeled-entry can be implemented by a virtual slot giving access to the slot `text` of the embedded label.
- The ones that propagate a property to some components of the new class. This is frequent in composite widgets, since we generally want that the decoration elements of the composite be affected in the same way as the main elements. For instance, changing the background color of a `labeled-entry` implies changing the color of the entry as well as its container and associated label. Slot propagation is generally implemented by the means of a virtual slot.

A possible implementation for the `labeled-entry` class could thus look as follows:

```
(class labeled-entry::entry
   (box (default #unspecified))
   (label (default #unspecified))
   ;; the container relief
   (relief
    (get (lambda (o)
           (box-relief (labeled-entry-box o))))
    (set (lambda (o v)
           (label-relief-set! (labeled-entry-label o) 'none)
           (box-relief-set! (labeled-entry-box o) v))))
   ;; the text of the label
   (title
    (get (lambda (o) (label-text (labeled-entry-label o))))
    (set (lambda (o v) (label-text-set! (labeled-entry-label o) v)))))
```

However, defining a new class is not sufficient to define a new widget. In effect, the protocol used by the Biglook library tells us that a widget must be first *realized* before being properly initialized. Widget realization is ensured by the `realize-widget` generic function. This function is generally overloaded by a new method for each widget class since widget realization depends on the type of the widget that must be created. This is the place where graphical tuning for the composite widget can be implemented.

For the `labeled-entry` class, in order to properly initialize its instances, we override `realize-widget` with the following method.

```
(define-method (realize-widget o::labeled-entry)
  (with-access::labeled-entry (parent label box)
     (set! box (instantiate::box
                  (parent parent)
                  (orientation 'horizontal)))
     ;; allocate the label
     (set! label (instantiate::label (parent box)))
     ;; invoke the entry realization
     (call-next-method)
     ;; re-parent the entry
     (set! parent box)))
```

This method first allocates a container and a decoration `label` before calling `call-next-method` in order to ensure a proper realization of the `entry`. Note that `label` reuses the container of the `entry` to be effectively *embedded* in the `labeled-entry`.

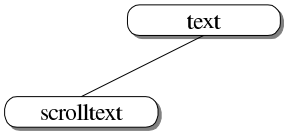
3.8 Reflection on widgets composition and generic functions

A generic function is a bundle of methods. Object behaviors are associated with generic functions instead of being associated with class definitions. Behavior specification is thus orthogonal to classes definition. Generic functions enable behaviors to be associated with instances after they have been created. When methods are defined inside classes definition, the behaviors of the instances are specified once and for all. Generic functions enable new behaviors to be specified anytime, anyplace. Generic functions permit user customizations that are hardly possible if methods are defined within classes. Of course, this customization may yield to situations where adding an erroneous method breaks existing codes because adding a method may change the treatment applied to already existing data structures. However we are convinced that this inconvenient is minor compared with the advantages of the extensibility capacity discussed in the next sections.

3.8.1 Extending public widgets

In this section we present how user applications may specify new behaviors for standard widgets. We present four scenarios: *1)* one based on generic functions *2)* another one using multiple inheritance and subtyping *3)* another one using single inheritance and multiple subtyping and *4)* a last one using parametric types.

Biglook proposes several kinds of text widgets. Let us concentrate on two simple ones: plain text (`text`) and text provided with scrollbars (`scrolltext`). `Scrolltext` is a composite widget whose implementation is very close to the one of `labeled-entry` presented Section 3.7. The scrollbar is the decoration of the `scrolltext`. Consequently, `text` and `scrolltext` are related by subtyping *and* inheritance relationships:

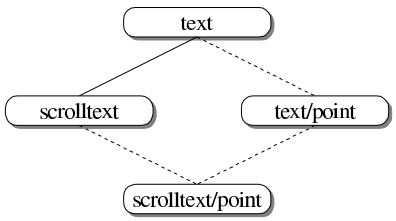


Each text widget has a cursor whose location is defined by two numbers: the line number and the column number. A user application may find to be more convenient to represent the cursor position as Emacs does, that is, with one unique number which is an integer representing the offset of the character from the start of the buffer. Switching from an $X \times Y$ positioning to a linear positioning and *vice versa* is algorithmically easy.

1. In order to implement linear positioning, provided with generic functions, a user application could simply define:

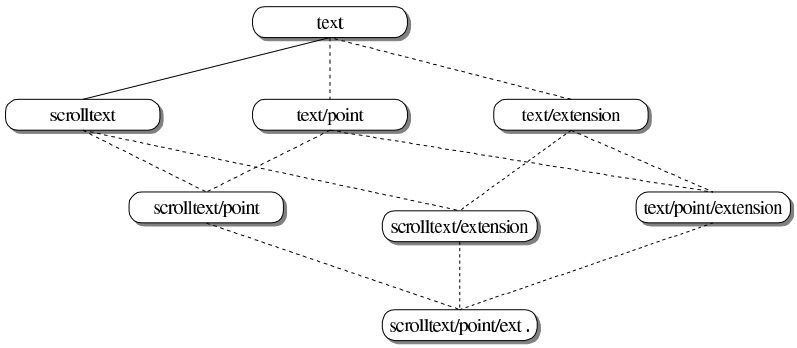
```
(define-generic (point::long t::text)
  (with-access::text t (cursor-x cursor-y line-width)
     ;; this is an over simplified implementation because in
     ;; practice not all lines are of same width
     (+ cursor-x (* line-width cursor-y))))
```

   This definition is valid for `text` and all its subclasses. It is thus valid for `scrolltext`. Because `text` and `scrolltext` share the same implementation for `point`, a regular Scheme function could be used instead of a generic function. However, one may imagine situations requiring specific behavior and thus different methods. Without generic functions three solutions can be deployed.
2. If the language supplies multiple inheritance such as C++ or O'Caml (Rémy & Vouillon, 1997)⋆, a user code may subclass `text` into a `text/point` class that will be provided with an extra `point` member function and will derive a new `scrolltext/point` inheriting from `text/point` and `scrolltext`:

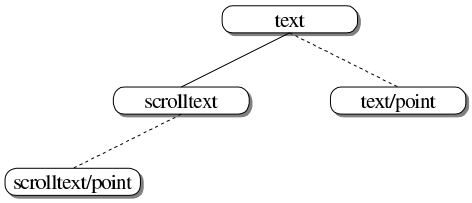


   If class declarations can be used, this solution is more complex than generic functions because *two* new classes have to be introduced for each new facility. If we want to add a second extension, we might have to introduce two additional classes and so on. This conducts rather rapidly to impractical inheritance schemes as shown below:

⋆ One should note that because O'Caml supports parametric polymorphism, the `point` facility could be implemented using a plain function. This solution would avoid building new classes but it is not as powerful as generic functions because a plain function cannot be overridden.



3. If the language does not provide multiple inheritance or offers only limited multiple inheritance such as Java, then adding our `point` facility in user applications is hardly possible. In Java, because `text` and `scrolltext` must be plain classes (by contrast to *interfaces*), it is impossible to define a class inheriting from both `text/point` and `scrolltext`. Thus a user application must use the inheritance tree:



   `Text/point` and `scrolltext/point` are no more in a subtyping relationship which means that an instance of `scrolltext/point` cannot be used when an instance of `text/point` is expected. Obviously, this is not as convenient and powerful as the generic function based solution.
4. Parametric types such as C++'s templates (Stroustrup, 1994) or MzScheme mixins (Flatt *et al.*, 1998) are a means to delay the concrete realization of classes. Parametric types help with program extensibility because actual classes are created in two steps: *i)* definition of the parametric class *ii)* effective realization of the concrete class. During step *ii* one may specialize concrete classes and inheritance relationships. Because parametric types allow to generate classes, in many situations they can be used in place of multiple inheritance. However, extending the previous text example using parametric types would lead to a solution close to the one based on multiple inheritance. The restrictions of the multiple inheritance also applies to parametric types. In addition, as stated in the paper (Findler & Flatt, 1998), parametric types require careful design in order to enable extensibility. Failing to conform to strict design rules compromises further extensibility.

3.8.2 Extending embedded widgets

When methods are defined in the class, adding a new service requires at least the declaration of a new extending class. However, this framework does not apply for already existing instances. This situation occurs frequently in object oriented li-

braries. For instance, the Biglook library proposes the `editor` class which is defined as:

```
(class editor::frame
   title::string
   window::frame
   minibuffer::entry
   ...)
```

Because most of the time, `editor` frames are used to implement text editors, the Biglook library offers the convenient `make-editor` function that instantiates an `editor` frame and fills it with a text widget and a set of default values for the other slots.

```
(define (make-editor title::string)
    (instantiate::editor
       (title title)
       (window (instantiate::text))
       ...))
```

In this case, `text` instances used to fill the `window` slots are instantiated by the `make-editor` constructor and users have no means to control over their creation since it is done by a library function. With such a restricted design for `make-editor`, the type of the `window` slot cannot be changed. The constructor `make-editor` lacks a parametric type and uses no inheritance. It is impossible for the user to add a facility such as emacs positioning to the embedded `text` windows, using the previously described subclasses framework of Section 3.8.1.

Because generic function are unrelated to class definitions, they can be used to extend already created instances. In our `editor` example, a generic function could extend the facilities provided by the embedded editor `window`. More arguments in favor of generic functions may be found in papers by C. Chambers (Chambers, 1998; Millstein & Chambers, 1999)).

4 Implementing Biglook

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

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.

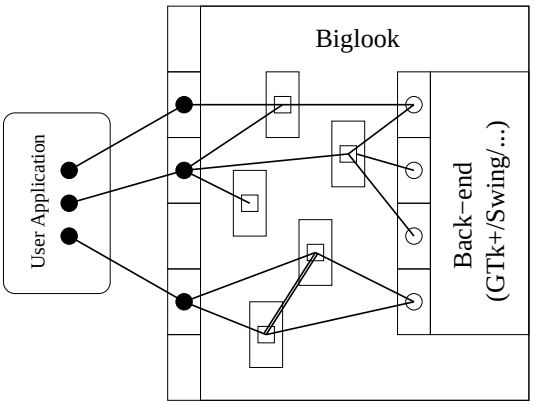


Fig. 9: The Biglook software architecture

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). Figure 9 shows the underlying architecture of the Biglook toolkit. In this figure, black circles stand for user objects, white squares stand for Biglook internal objects, and white circles stand for proxies pointing to back-end objects.

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-widget)
              (<builtin-label-text> builtin-widget))))
    (set (lambda (o::label v::procedure)
           (with-access::label o (builtin-widget)
              (<builtin-label-text-set!> builtin-widget 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. The memory layout for `label` instances is presented in Figure 10.

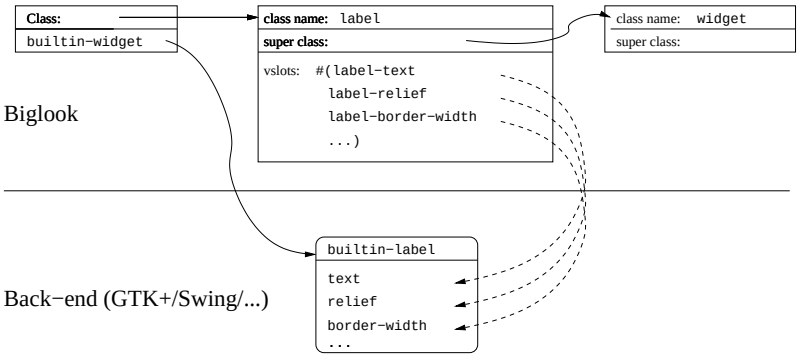


Fig. 10. `Label` instances memory layout.

Now that the slots of a label widget are defined, we must define how such a widget has to be initialized. As with composite widgets, 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-widget parent)
      (set! builtin-widget (<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-widget` slot. This slot creates the link between the Biglook toolkit and the back-end, as already seen in Figure 10.

4.3 Reflection on virtual slots

It is tedious to implement a GUI because it needs a lot of graphical tuning that requires successive endeavors. Helping GUI programming is the task of *interface builders*. These tools rely themselves on GUI. They enable the available widgets of the toolkits to be graphically configured and inserted in the user interfaces. Interface builders must be aware of the whole available widgets and the whole configuration options of these widgets. If the toolkit implementation language is provided with introspection and if widget configurations are implemented by the means of class slots, builders can rely on it to present configuration panel for widgets (pre-defined widgets as well as widgets defined in user programs). In the case of Biglook, all the widget configurations are implemented by the means of virtual slots. So, introspecting the class of a widget enables its full tuning (Superina, 2000).

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 (Gallesio, 1995) 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 (Dybvig, 1998). 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 (Flatt *et al.*, 1999), a part of the DrScheme project (Felleisen *et al.*, 1998), 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 (Smart, 1992), a toolkit that is available under various platforms (Unix, Windows, etc.). As STk, this toolkit is completely dependent of its back-end.

Interested readers can find performance evaluation of these implementations in (Gallesio & Serrano, 2001).

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 (Weis & *al.*, 1991).

The first attempt, CamlTk, is quickly surveyed in (Rouaix, 1996). 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 (Furuse & Garrigue, 1995). 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 (Noble & Runciman, 1994; Vullings *et al.*, 1995). Several solutions have been proposed: Fudget by Carlsson and Hallgren (Carlsson & Hallgren, 1993; Carlsson & Hallgren, 1998), Haggis by Finne and Peyton

Jones (Finne & Peyton Jones, 1995), TkGofer by Vullings and Claessen (Claessen *et al.*, 1997) and, more recently the extension of the former TclHaskell library: FranTk by Sage (Sage, 2000).

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

Bobrow, D., DeMichiel, L., Gabriel, R., Keene, S., Kiczales, G., & Moon, D. 1988 (Sept.). Common lisp object system specification. *special issue*. SIGPLAN Notices, no. 23.

Carlsson, M., & Hallgren, T. (1993). FUDGETS - A Graphical User Interface in a Lazy Functional Language. *Pages 321–330 of: Fpca '93 - conference on functional programming languages and computer architecture*. ACM Press.

Carlsson, M., & Hallgren, T. 1998 (March). *Fudgets — Purely Functional Processes with applications to Graphical User Interfaces*. Ph.D. thesis, Department of Computing Science, Chalmers University of Technology, S-412 96 Gteborg, Sweden.

Chambers, C. 1993 (Mar.). *The Cecil Language: Specification and Rationale*. Tech. rept. 93-03-05. University of Washington, Department of computer Science and Engineering.

Chambers, C. (1998). Towards Diesel, a Next-Generation OO Language after Cecil. *Int'l workshop on foundations of object-oriented languages*. invited talk.

Claessen, K., Vullinghs, T., & Meijer, E. (1997). Structuring Graphical Paradigm in TkGofer. *Int'l conf. on functional programming*.

Dalheimer, M. (1999). *Programming with Qt*. 1st edn. O'Reilly.

Dybvig, K. (1998). *Chez Scheme User's Guide*. Cadence Research Systems.

Felleisen, M., Findler, R., Flatt, M., & Krishnamurthi, S. (1998). The DrScheme Project: An Overview. SIGPLAN *notices*.

Findler, R., & Flatt, M. 1998 (Sept.). Modular Object-Oriented Programming With Units and Mixins. *Int'l conf. on functional programming*.

Finne, S., & Peyton Jones, S. 1995 (Sept.). Composing Haggis. *Fifth eurographics workshop on programming paradigm for computer graphics*.

Flatt, M., Krishnamurthi, S., & Felleisen, M. 1998 (Jan.). Classes and mixins. *Pages 171–183 of: Symposium on principles of programming languages*.

Flatt, M., Findler, R., Krishnamuryhi, S., & Felleisen, M. (1999). Programming Languages as Operating Systems (*or* Revenge of the Son of the Lisp Machine). *Int'l conf. on functional programming.*

Furuse, J., & Garrigue, J. 1995 (Oct.). *A label-selective lambda-calculus with optional arguments and its compilation method.* Tech. rept. RIMS Preprint 1041. Research Institute for Mathematical Sciences, Kyoto University.

Gabriel, R. (1991). *Lisp: Good News, Bad News, How to Win Big.* http://www.ai.mit.edu-/articles/good-news/good-news.html.

Gallesio, E. 1995 (July). *STk Reference Manual.* Tech. rept. RT 95-31a. I3S-CNRS/Univ. of Nice–Sophia Antipolis.

Gallesio, E. (1996). Designing a Meta Object Protocol to Wrap a Standard Graphical Toolkit. *Isotas.*

Gallesio, E., & Serrano, M. 2001 (Sept.). *Graphical user interfaces with biglook.* Tech. rept. I3S/RR–2001-13–FR. I3S-CNRS/Univ. of Nice–Sophia Antipolis.

Goldberg, A., & Robson, D. (1983). *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley.

Gosling, J., Yellin, F., & the Java Team. (1996a). *The Java Application Programming Interface, Volume 2: Window Toolkit and Applets.* Addison-Wesley.

Gosling, J., Joy, B., & Steele, G. (1996b). *The Java Language Specification.* Addison-Wesley.

ISO/IEC. 1990 (July). *9899 programming languages – C.* Tech. rept. DIS 9899. ISO.

Kelsey, R., Clinger, W., & Rees, J. (1998). The Revised(5) Report on the Algorithmic Language Scheme. *Higher-order and symbolic computation*, **11**(1).

Kiczales, G., des Rivières, J., & Bobrow, D. (1992). *The Art of the Metaobject Protocol.* Cambridge, Mass., USA: MIT Press.

Millstein, T., & Chambers, C. 1999 (June). Modular Statically Typed Multimethods. *Pages 279–303 of: European conference on object-oriented programming.*

Noble, R., & Runciman, C. 1994 (Feb.). *Functional Languages and Graphical User Interfaces – a review and a case study.* Tech. rept. 94-223. Department of computer Science, University of York.

Nordlander, J. (1999). *Reactive Objects and Functional Programming.* Ph.D. thesis, Department of Computing Science, Chalmers University of Technology.

Ousterhout, J. (1994). *Tcl and the Tk toolkit.* Addison-Wesley.

Pennington, H. (1999). *Gtk+/Gnome Application Development.* New Riders Publishing.

Queinnec, C. (1993). Designing MEROON v3. *Workshop on object-oriented programming in lisp.*

Rémy, D., & Vouillon, J. (1997). Obective ML: A simple object-oriented extension of ML. *Pages 40–53 of: Symposium on principles of programming languages.*

Rouaix, F. (1996). A Web navigator with applets in Caml. *Pages 1365–1371 of: Proceedings of the 5th international World Wide Web conference, in Computer Networks and Telecommunications Networking*, vol. 28:7–11. Elsevier.

Sage, M. 2000 (Sept.). FranTk – A declarative GUI language for Haskell. *Int'l conf. on functional programming.*

Serrano, M. 1994 (Dec.). *Bigloo user's manual.* RT 0169. INRIA-Rocquencourt, France.

Serrano, M. 1999 (June). Wide classes. *Pages 391–415 of: Proceedings ecoop'99.*

Shalit, A. (1996). *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language.* Addison-Wesley.

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

Stroustrup, B. (1994). *The Design and Evolution of C++*. Addison-Wesley.

Superina, M. 2000 (Jan.). Buildoo. *Actes des journées jfla*.

Vullings, T., Tuijnman, D., & Schulte, V. (1995). Lightweight GUIs for Functional Programming. *Int. symp. on programming languages, implementations, logics, and programs*.

Wadler, P. (1998). Why no one uses functional languages. SIGPLAN *notices*, **33**(8), 23–27.

Walrath, K., & Campione, M. (1999). *The JFC Swing Tutorial: A Guide to Constructing GUIS*. Addison-Wesley.

Weis, P., & *al.* (1991). *The CAML Reference manual*. Tech. rept. 121. INRIA-Rocquencourt.