Thinking Scheme

Kenneth A Dickey

Ken.Dickey@whidbey.com

Abstract

Like APL crossword puzzles, the goal in Scheme is not how to find ***a*** way to implement something but to discover the best implementation style(s) to adopt based on the computational patterns inherent in the artifact being developed.

This paper describes the author's experience building GUI-Toy, a simple graphical user interface, using SDL, two different object systems, and multiple Scheme implementations.

Keywords Graphical User Interface, Scheme Programming Language, Software Development, Object System, Foreign Function Interface

1. Introduction

Computing systems today typically suffer from excess complexity. Some of this complexity is inherent, but much is simply the brambles of history as people cycle through projects and knowledge is lost. One way to understand software systems is to build them. Developing software allows one to understand and challenge conventional wisdom and clarify one's thinking. [Re]implementing theory leads to comprehension.

Like APL [1] crossword puzzles, the problem in Scheme [25] is not how to find ***a*** way to implement something but to discover which implementation style(s) to adopt based on the patterns in the computational artifact being developed. An analogy is in writing poetry verses prose. In poetry, one looks for and coins new metaphors which make use of existing mechanisms of association to compactly communicate meaning. Programming languages with a high impedance get in the way by requiring one to write verbose prose.

This paper describes the author's experience building GUI-Toy, a simple graphical user interface, using SDL [26], two object systems with different object models, and multiple Scheme implementations. To comprehend computational systems, it is often still useful to walk down well trodden paths. Transliterating ideas from other languages into Scheme gives a common baseline for comparison and contrast.

GUI-Toy [13] is a proof of concept for a graphical user interface in the spirit of Sun's Lively Kernel [20] or Squeak Smalltalk [28] and indeed grew out of a desire to understand the Lively Kernel code. It consists of two computational engines, the rendering of SDL and the object structure and event processing of Scheme. These two engines are implemented variously as separate threads or as separate processes. The main window has a bitmapped background, nested graphic objects with mouse dragging, and simple time based animation. Events are targeted to graphic objects based on Z-order and mouse sensitivity.

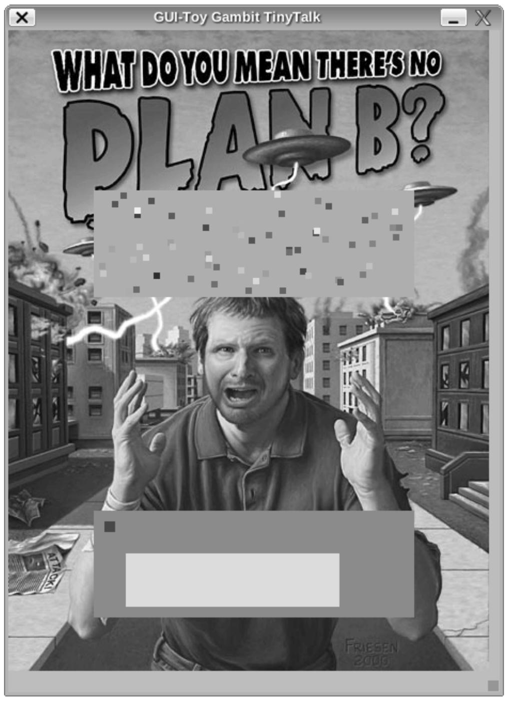

Figure 1. GUI-Toy window.

2008 Workshop on Scheme and Functional Programming 20 September 2008, Victoria, British Columbia.

The two object systems used are *TinyTalk* [29] and *OOPS* [21]. While both object systems support multiple inheritance, they are diametrically opposed in philosophy and implementation.

OOPS is a sophisticated object system for Gambit Scheme [11] in the legacy of CLOS [9] and Dylan [10]. OOPS supports generic functions, predicate and limited types, support for Scheme native types via virtual classes, condition handlers with restarts, access to Gambit's native record types, and a substantial collection library, including thread-safe hash tables which allow concurrent access by multiple threads. Classes are data structures which are also the procedures used to create instances. It could be ported to Scheme implementations which support access to closure environments such as Larceny [18].

TinyTalk is a minimalist object toolkit with first class objects, selector name lookup independent of the Scheme namespace, and no global tables. Unlike OOPS, TinyTalk uses single selector dispatch and does not note or use the types of its arguments. It uses a prototype/delegation model and was implemented in less than 500 lines of code. This is about the size of the OOPS define-class macro alone. It is quite portable.

2. GUI Architecture

The model presented to the user is that of nested graphical objects with a Z-order (objects are always in front of or behind others). Objects may be moved within their container. Moving a container moves its contents as well. I.e. contents are always relative to their container.

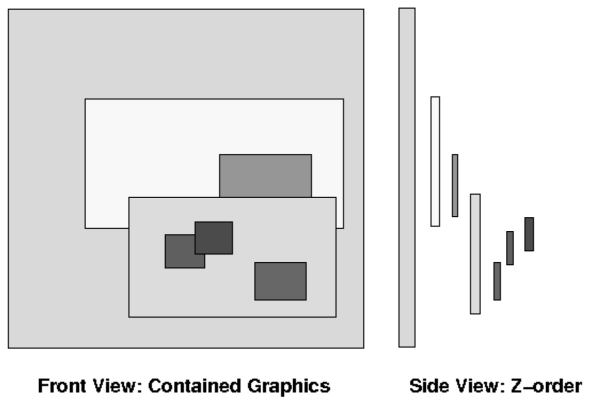

Figure 2. Contained Rectangles and Z-Order.

Figure 2 shows a root-graphic containing two rectangles, the frontmost of which contains three rectangles.

In Gui-Toy there is a crisp separation between the graphical rendering system and the computational objects organizing their appearance and behavior. In the implementation the SDL-interface, graphical objects implementation, and the composition of the two are maintained as three separate files. The computational flow is of drawing commands from Scheme to SDL and user events from SDL to Scheme.

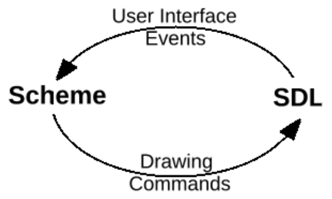

Figure 3. User Events and Drawing Commands

There are four general categories of events:

- **mouse events** move, button-down and button-up are directed to the *mouse-target* object.
- **keyboard events** are directed to the *keyboard-focus* object.
- **focus-change events** (mouse-focus, keyboard-focus, application-focus) are directed to the *application-focus* object.
- **timer-tick events** are given to the *tick-announcer* object about 30 times a second, which drives the frame-rate for redisplay and for timer-based animation.

A *mouse target* is the topmost mouse sensitive graphic at the current mouse position. Finding the mouse target when a mouse event occurs involves searching. The search starts at the frontmost content of the rearmost graphic, the *root graphic*. The general strategy is to check contained objects in inverse Z-order, from front to back. If an object contains the mouse point, then its contents are searched in the same fashion recursively. This strategy is modified somewhat based on mouse sensitivity, which in turn is based on experience with Apple's SK8 (pronounced "skate") multimedia development environment [27].

When *mouse-sensitivity* is

- **normal** - the mouse target is content or self
- **transparent** - the mouse target is content or container, never self
- **opaque** - the mouse target is always self
- **invisible** - the mouse target is always container

Each sensitive mouse-target object decides itself how to handle mouse events, which may indeed be ignored.

In keeping with the desire for ease of comprehension, the code is kept as simple as possible. For example, GUI-Toy does **not** track *damage areas*. The basic idea of damage areas (a.k.a. damage rectangles) is that screen changes are expensive and other computation is cheap, so only areas that have changed (are 'damaged') need to be redrawn. GUI-Toy simply keeps two buffers, drawing all graphics on the "off screen" buffer from back to front and then swapping buffers between screen redraws. This has sufficed in tests with several hundred small animated objects. It will be interesting to find the knee of the performance curve, if it exists, where damage rectangles will begin to pay for their added complexity.

Renderers are another idea from SK8. A renderer encapsulates the visual presentation of a shape, i.e. color, gradient, pattern/stipple, picture. This is separate from the graphic shape, which may be algorithmic (rect, polyline, etc.) or an arbitrary bitmask. A renderer takes a shape, which has a position relative to its container, a surface, and a point representing the container's screen position in global coordinates.

The method for rendering is now dirt simple [from the TinyTalk code for graphic objects]:

```
[(display self screen parent-offset)
; Convert from local to global coordinates.
 (let ((my-global-offset
          [$ + parent-offset
             [$ top-left-point self]]))
  (when [$ visible? self]
     ; paint the pixels
     ([$ renderer self]
         self screen my-global-offset)
     ; tell contents to display
     (for-each
      (lambda (graphic)
        [$ display graphic
             screen my-global-offset])
      [$ contents self])
   ) )
]
```

Given the above, the primary features of a GUI-Toy graphic object are a shape, a renderer, container and contents, a mouse-sensitivity, and event handlers.

3. Object System Comparison

After a programming hiatus, the author noted the publication of R6RS [25]. The TinyTalk object system was developed in order to learn and explore the differences between the new and previous reports. TinyTalk was then ported to Gambit using old style macros and GUI toy started using Gambit's foreign function interface, timer events and threads. Once GUI-Toy was running, it was natural to transliterate the code into the more CLOS-ish style of OOPS to compare coding styles.

OOPS grew out of reading a number of object dispatch theory papers (e.g. [5]) and a desire to better understand the design trade-offs in supporting sophisticated and efficient generic function dispatch. OOPS includes support for singletons, type unions, one-of types, limited types, type predicates, et cetera, while striving to be as simple as possible. The current implementation of generic dispatch lazily builds tail-recursive lookup automata [7] [31] which use an efficient subtype test [2].

TinyTalk, on the other hand, was designed to be very flexible in order to explore alternative object oriented styles before committing to a particular, optimized OO system. Its design is consciously minimalist. It is prototype based and uses single argument dispatch.

On the surface, usage of TinyTalk and OOPS is fairly similar. There are, however, subtle differences which point to some of the deep differences ”under the hood”.

We will explore this briefly with an example of point objects, which are used in GUI-Toy to represent screen pixel positions.

In OOPS, points look a lot like in CLOS:

```
(define-class <point> ()
  ( [x init-value: 0  type: <s32int>]
    [y init-value: 0  type: <s32int>]
  )
)

(define-method (->string (p <point>))
  (format "#<point x:~a y:~a>"
          [x p] [y p]))

(define p (<point> x: 10 y: 22))
```

In TinyTalk one can do this in several ways.

One can make an anonymous point and add methods to it.

```
(define p (object ( [x 10] [y 22] ) ))

[$ add-method! p 'point? (lambda (self) #t)]

(define-predicate point?)

[$ add-method! p '->string
   (lambda (self)
     (format "#<point x: ~a y: ~a>"
             [$ x self] [$ y self]))
]
; ...
```

The $ character is a macro which just gives the ”selector” to the lookup function along with the first argument and returns either a method or a suitable error procedure which is then applied.

```
;; [$ <selector-sym> <obj> <arg> ...]
(define-syntax $  ; send [user syntax]
 (syntax-rules ()
  [($ <selector> <obj> <arg> ...)
  ;=>
   ((-> '<selector> <obj>) <obj> <arg> ...)
  ]))
```

In this style a point is a *prototype*.

To make new points, one clones the prototype. Adding methods to the point p adds methods only to the single object p.

```
(define p2 [$ deep-clone p])
```

Alternatively, one can define a constructor function.

```
(define (new-point x y)
  (object ( [x x] [y y] )
     [(point self) #t)]
     [(->string self)
      (format "#<point x: ~a y: ~a>"
              [$ x self] [$ y self])]
     ; ...
) )
```

While the above looks like a class definition, it should be noted that in the implementation, each point gets a new set of closures for point methods. As instance value accesses are done via getter functions, one can factor them out to get a more class-based style.

```
(define proto-point
  (object () ; Nota Bene: methods only
     [(point self) #t)]
     [(->string self)
      (format "#<point x: ~a y: ~a>"
              [$ x self] [$ y self])]
     ; ...
) )

(define (new-point x y)
  ( [x x] [y y] ) ; NB: data only
  [(delegate self) proto-point]
)
```

In this style, a new point has its own x and y accessors, but shares the proto-point methods. One can now add methods either to a single instance, or add them to the shared proto-point, so that all point instances will share the behavior.

An additional flexibility comes from using selectors. As method and instance data names are local to objects and not part of the Scheme namespace, one can easily reuse names like **+** without worrying about interactions with existing definitions.

OOPS by contrast has a great deal of convenience built into its relativity Procrustean style. For example, it is very easy to add variant methods to OOPS because of the behind the scenes centralized information.

OOPS:

```
(define-method (add (p1 <point>)(p2 <point>))
  (<point> x: (+ [x p1] [x p2])
           y: (+ [y p1] [y p2])))

(define-method (add (p <point>)(n <integer>))
  (<point> x: (+ [x p] n)
           y: (+ [y p] n)))

(define-method (add (n <integer>)(p <point>))
  (<point> x: (+ n [x p])
           y: (+ n [y p])))
```

Doing the same in TinyTalk requires dealing with distributed information.
TinyTalk:

```
(define proto-point
 (object () ; methods only
  [(point? self) #t]
  [(add self other)
   (cond
    ((point?  other)
     (new-point (+ [$ x self] [$ x other])
                (+ [$ y self] [$ y other])))
    ((number? other)
      (new-point (+ [$ x self] other)
                 (+ [$ y self] other)))
    (else
       (error 'point:add
            "Can't add self to other"
             self other))) ]
  ; ...
) )

[$ add-method!
; get deputy prototype for built-in type
 (deputy-object 3) ;-> integer prototype
 'add  ; method name
 (lambda (self other)
  (cond
   [(point? other)
    (new-point (+ self [$ x other])
               (+ self [$ y other]))]
   [(number? other) (+ self other)]
   [else
     (error 'number:add
       "Can't add self to other"
       self other)]
))]
```

As one might suspect, TinyTalk is a language for "consenting adults". Its flexibility comes at the price of having to maintain a clear idea of one's intent and in having to write code specifics in more detail. For example, differences between shallow and deep cloned objects and the ability to attach methods to individual objects is sometimes surprising in practice. One case where this matters is in cloning of graphic objects. A graphic objects has the shape (x, y, width, height) as a delegate. If a shallow-clone is done, the shape is shared between both the original and the cloned graphic. When one is moved or resized, the other is moved and resized as well. Since one is on top of the other, the second is hidden and it appears that it has vanished. Deep-cloning gives each graphic its own shape and they can be moved and resized independently.

OOPS is fairly classic to one familiar with CLOS or Dylan. Notable features of OOPS include *predicate dispatch types*, *limited types*, *one-of types*, and the use of *virtual classes* to support native Scheme value types.

```
(define-virtual-class <string> <indexed>
  ((element-type type: <type>
                 init-value: <character>
                 allocation: each-subclass:)
   (fill         type: <character>
                 init-value: #\space
                 allocation: override:))
  ; replace inst w string
  (lambda (inst)
     (make-string (size inst) (fill inst))))

;E.g.
 (is-a? "abc" <string>)   ;--> #t
 (class-of "abc")         ;--> <string>
 (<string> size: 3 fill: #\x)  ;--> "xxx"

(define <even?>
  (<predicate-type> superclass: <integer>
                    test-for: even?))

(define <u8int> ;  Useful in a FFI
  (<limited-range> superclass: <integer>
                   min-value: 0
                   max-value: #xFF))

(define <vowels>
  (apply <one-of>
      (string->list "aeiouAEIOU")))
```

Given this flexibility in dispatch mechanics, it is unsurprising that argument type checking is delegated to the generic dispatch system in the OOPS rendition of the GUI-Toy code.

```
(define <mouse-sensitivity>
  (<one-of>
   'normal 'invisible 'transparent 'opaque))

(define-class <graphic> ()
  ([shape     type: <shape>]
   [renderer  type: <renderer>]
   [mouse-sensitivity
              type: <mouse-sensitivity>
              init-value: 'normal]
   [visible?  type: <boolean> init-value: #t]
   ; Container is #f or a <graphic>
   [container]
   ; Contents are z-ordered back to front
   [contents  type: <list>  init-value: '()]
   ; Delegate location to SHAPE
   [x
    allocation: virtual:
    slot-ref:  (lambda (g)   (x (shape g)))
    slot-set!:
        (lambda (g v) (x-set! (shape g) v))
    ]
   ; ... ... ...
   ; Individual instances must override
   ;  default handlers to do something useful.
   [mouse-move-handler type: <function>
           init-value: ignore-graphic-event]
   ; ... ... ...
; Instance Init:
(lambda (me)
 (when (container me)
 ; a <graphic> unless the root-graphic
  (let* ( [my-container (container me)]
          [my-siblings
            (contents my-container)] )
 ; Contents list is 1st in Z-order
  (contents-set! my-container
     (concatenate my-siblings (list me)))
))))
```

3.1 OO Style Contrast

In contrasting the styles, one finds OOPS comforting. Keywords, while verbose, provide additional documentation. Given academic emphasis on type theory, type based dispatch, aside from complexities on class precedence list calculation, is fairly intuitive.

TinyTalk, on the other hand, allows one to build exactly what is wanted, but its minimalism is sometimes disconcerting. For example, the access functions act as both setters and getters. This violates the user interface guideline that things which are different should *look* different.

```
[$ x point]   ; get value for x
[$ x point 3] ; set value of x to 3
```

In OOPS this is more distinguishable.

```
[x point]         ; get value for x
[x-set! point 3] ; set value of x to 3
```

On the other hand, it is very handy for TinyTalk not to add to the Scheme name space. It is often convenient to have a variable with the same name as a method.

```
(let ( [shape [$ shape graphic]] )
; Here one can use variable shape as well
;  as calling method [$ shape foo]
)
```

The contrast in object styles has been interesting. In OOPS one uses generic functions which collect all methods of the same name and are separate from the data objects which they use in object dispatch. The generic function name is required to be in the Scheme namespace. In TinyTalk the *selector* symbol is given to the data object, along with any other arguments, and the first object argument determines which method to invoke. TinyTalk method names are unknown to the Scheme namespace.

In this small example, the author found these object system differences as simple matters of style. Neither style of object system was a clear win in all cases. As a user, OOPS code seems more readable, but as an implementor it is much more work to understand and implement and requires more runtime internals knowledge and support. TinyTalk code requires a bit more care to write, but the TinyTalk implementation is quite small, comprehensible, and ports easily. On balance, it seems that in the case of GUI-Toy the OOPS system does not yet carry the weight of its complex implementation.

Where the difference matters is in support of GUI usability. The SK8 experience was that multi-media authors who were not programmers liked to work from pallets of media objects which they could clone and then add behaviors (methods) to. The authors tend to make large numbers of idiosyncratic objects and sometimes move or clone behaviors as well. As one scales up GUI features, the prototype implementation of objects fits better with this style of media authoring.

4. Foreign Function Interface Issues

Unlike Common Lisp's CFFI [6], Scheme has no standard foreign function interface. Due to the brambles of various Scheme implementations, several styles have evolved with some interesting differences (e.g. [19] [23] [12] [4] [8]). This makes porting and testing any non-trivial graphic UI a significant project.

Perhaps the most portable way to access non-Scheme code is to avoid using a FFI in favor of pipes to a server process as used in PS/Tk [24], a portable interface between a number of Scheme implementations and the Tk Toolkit. The trade-off here is that one must deal with multi-threading issues between a Scheme implementation and the server and keep interface encodings consistent. The obvious way to portably support a FFI of any complexity is to define a declarative interface and generate the C glue code. As the size of the interface increases, a compression strategy will probably be required. A "portable FFI" would help evolution here as well.

Starting with the simple server in the Worms game developed in Ikarus Scheme [16], the author implemented an SDL interface and server which currently works across Chez, Gambit, Ikarus, Larceny, PLT, and Ypsilon Schemes. The use of a server process does allow for rapid porting. While simple, the C server code required considerable debugging time and the author was forced to relearn why he had forgotten the C language.

To give a brief feel for this interface, some of this mechanism follows.

A request for an operation on the SDL interface writes binary data in a format known by the server.

```
(define SDL::load-bmp-file
 (lambda (file-name-string)
  (unless (string? file-name-string)
       (report-arg-check-error
              'SDL::load-bmp-file
              'file-name-string
              'string?))

 (let ( [return-id (next-return-id)] )
   (put-Uint8 port->sdl 33) ; call tag
   (put-Uint16 port->sdl return-id)
   (put-Str port->sdl file-name-string)
   (flush-output-port port->sdl)
   (await-return return-id))
))
```

The server processes the command and writes to a text port read by the Scheme system. The convention used is that events are lists with a leading symbol which denotes what event is taking place. For example:

```
(sdl-result 2 134574568)
(focus-change 0 1)
(focus-change 1 1)
(key-down 115 0 115)
```

The Scheme code is event driven with a simple dispatch function.

```
(define (client-event-loop)
  (process-sdl-event
    (SDL::get-event 'wait))
  (unless sdl-to-exit?
    (client-event-loop)))

(define (process-sdl-event evt)
; ...
(case (car evt)
 [(sdl-result)
  (SDL::handle-return-event evt)]
 [(mouse-move)
  (%handle-mouse-event
           (evt->mouse-move evt))]
 [(tick)
  [$ announce tick-announcer
             (make-event 'tick)]]
 ; ...
 ])
)
```

4.1 Interleaved Event Processing

One interesting problem with a pipe interface to a subprocess is that processing of asynchronous events (mouse move, timer ticks, keyboard input) is interleaved with the processing of commands which return values.

Fortunately, Scheme's call-with-current-continuation function provides a ready mechanism with which to associate return values with computations awaiting results. The SDL interface and GUI code are separate, but the GUI registers the client-event-loop with the SDL interface and gives sdl-return events back to the interface as shown above. Aside from this communication for execution control, association of results with SDL commands remains local to the SDL interface. This code uses the return-id to find and dispatch to the appropriate continuation.

The author started with the design of a more complex coroutine system but, in realizing that the piping mechanism acts as a concurrency constraint, eliminated the complex cases. After a day or two reviewing papers (e.g. [3]) and thinking about the problem, the implementation, testing, and refinement took under an hour. The implementation is small enough to be included here.

```
(define (await-return return-id)
 (call/cc
  (lambda (receiver)
   (register-return return-id receiver)
   (client-event-loop))))

(define (register-return id receiver)
  (set! return-cont-alist
       (alist-cons id receiver
                  return-cont-alist)))

(define (process-return id val)
  (let ([bucket
        (assq id return-cont-alist)])
    (unless bucket
      (error 'process-return
          "can't find return for id"
           id val))
    (set! return-cont-alist
       (remq bucket return-cont-alist))
    ; NB: continuation never returns
    ((cdr bucket) val)
) )

(define (SDL::handle-return-event evt)
  (let-values
    ( [(id result)
       (check-return-shape evt)] )
      (if id
        (process-return id result)
        (error 'SDL::handle-return-event
              "badly formed return event"
                                  evt)))))
```

4.2 Declarative Call Interface

The code to match Scheme ”calls” to SDL reads is tedious and care must be taken to match the number and interpretation of bytes sent by both sides of the pipe. This naturally led to the development of a simple, declarative FFI to generate both the Scheme and C call code to more easily add calls to the system.

The interface descriptions enable the code generator to insert basic checks and spread object fields to be gathered on the C side into stack allocated structs.

```
(describe-interface
;   ...
(describe-call
  (Scheme SDL::load-bmp-file
          (file-name-string char*))
  (C "SDL_LoadBMP"
     (file-name-string)
     (pointer "SDL_Surface")))

(describe-c-struct SDL_Rect
    (x Sint16) (y Sint16)
    (w Uint16) (h Uint16))

(describe-access (SDL_Rect rectangle?)
; (c-field-name scheme-getter-name) ...
  (x x) (y y) (w width) (h height))

(describe-call
 (Scheme SDL::draw-rect
       (surface (pointer "SDL_Surface"))
       (rect (access rectangle? SDL_Rect))
       (rgb-color Uint32))
 (C  "SDL_FillRect"
       (surface rect rgb-color) void))
;...
)
```

The generated Scheme code does basic error checks and reporting, puts out a shared *call tag*, and accesses object fields.

```
(define (SDL::draw-rect surface rect rgb-color)
  (unless (Uint32? surface)
     (report-arg-check-error
          'SDL::draw-rect 'surface 'Uint32?))
  (unless (rectangle? rect)
     (report-arg-check-error
          'SDL::draw-rect 'rect 'rectangle?))
  (unless (Uint32? rgb-color)
     (report-arg-check-error
          'SDL::draw-rect 'rgb-color 'Uint32?))

  (put-Uint8 port->sdl 34) ; call tag
```

```
   (put-Uint32 port->sdl surface)
   (put-Sint16 port->sdl [$ x rect])
   (put-Sint16 port->sdl [$ y rect])
   (put-Uint16 port->sdl [$ width rect])
   (put-Uint16 port->sdl [$ height rect])
   (put-Uint32 port->sdl rgb-color)
   (flush-output-port port->sdl)
)
```

The generated C code dispatches on the call tag, stack allocates temporaries, makes the SDL call, and returns a result as required.

```
case (char)(34):
{ /* SDL::draw-rect -> SDL_FillRect */
   unsigned char buf[16];
   read(fileno(stdin), buf, 16);
   int offset = 0 ;

   /* surface */
   Uint32 surface
          = getUint32( buf, offset ) ;
          offset += sizeof( Uint32 ) ;
   /* rect */
   SDL_Rect rect ;
   rect.x = getSint16(buf, offset) ;
          offset += sizeof( Sint16 ) ;
   rect.y = getSint16(buf, offset) ;
          offset += sizeof( Sint16 ) ;
   rect.w = getUint16(buf, offset) ;
          offset += sizeof( Uint16 ) ;
   rect.h = getUint16(buf, offset) ;
          offset += sizeof( Uint16 ) ;
   /* rgb-color */
   Uint32 rgb_color
          = getUint32( buf, offset ) ;
          offset += sizeof( Uint32 ) ;

   SDL_FillRect( (SDL_Surface*)(surface),
                 &(rect),
                 rgb_color );
}
break;
```

Previous to the implementation of the interface generator, the SDL interface was independent of the object system(s) used. In order to use argument checks and object accessors, the SDL interface currently imports the graphic object interface. This could be changed by creating a separate library which does the checking, spreads the arguments, and calls the (then) object system independent SDL interface. One could then use either the checked or unchecked interface.

4.3 Performance

Qualitatively, native compiled Scheme implementations easily handle 200 or more animated objects at 30 frames per second with live dragging. Interpreted Schemes fall off a bit earlier but typically handle 60 animated objects well.

A best case for the pipe interface is redrawing the screen objects. In this case, no result is required and the SDL rendering cost is charged against the SDL server process. Direct FFI calls include the SDL rendering cost.

OO	kind	60 graphics	200 graphics
OOPS	FFI	12 ms	24 ms
TinyTalk	FFI	12 ms	16 ms
TinyTalk	Pipe	4 ms	8 ms

. **Display Redraw** [Gambit, Linux, 1 GB RAM]

The worst case for the pipe implementation is when a result is required. In the following, a trivial procedure was called which adds two numbers and returns the result. In this case the pipe code has to capture the continuation, parse the result, associate the result with the initial continuation, and periodically garbage collect the parsed i/o data objects. The FFI on the other hand is in a tight loop and is not required to collect intermediate data.

Note that these timings are a qualitative sampling. Other activity is taking place and no attempt was made to optimize calculation or normalize results.

Scheme	kind	1000 Calls	1,000,000 Calls
Ikarus	Pipe	16 ms	18,909 ms
Ypsilon	Pipe	16 ms	26,158 ms
Ypsilon	FFI	4 ms	716 ms
PLT-r6rs	Pipe	160 ms	88,569 ms
PLT-r6rs	FFI	4 ms	452 ms
Gambit	Pipe	36 ms	48,862 ms
Gambit	FFI	2 ms	2,134 ms
Larceny	Pipe	116 ms	112,283 ms
Larceny	FFI	4 ms	1,856 ms

. **Call** with 2 args and **Return** result

5. Wins and Losses

There are two kinds of project time: *rapid progress* and *speed bumps*.

Let's start with the speed bumps. An amazing amount of time was wasted debugging C code and learning enough of LaTeX to produce this paper. Debugging old style macros (used in Gambit) was somewhat of a time sink. In porting GUI-Toy code to various Scheme systems, underdocumentation and large code bases were sometimes an impediment [notable exceptions were Chez Scheme, which ported in about an hour and PLT-r6rs which came up even faster]. There are certainly too many pre-R6 alias names for arithmetic-shift-right.

Wins included regression test suites for both object systems (during development the author completely changed the implementation of the OOPS access functions). Another was keeping the same SDL interface for both native FFI and the process/pipe implementation. Parnas' idea of hiding design decisions [22] is of note here. Despite the controversies surrounding R6RS and some oddities, such as requiring calls to initialize libraries, the author has found in porting GUI-Toy across a number of scheme systems that the R6RS systems presented the fewest speed bumps. E.g. there is one name for bitwise-arithmetic-shift-right !

6. Next Steps

This project has evolved in phases of exploration and consolidation with many side trips. There are two seeds here: a simple GUI and a portable FFI. With care and watering it is hoped they will nurture each other. The seed of the GUI is expected to grow to approximate the features of Lively or Squeak. The seed of a portable FFI was done in isolation but with community input could mature into or inspire the creation of a common cross-implementation Scheme foreign function call interface. A particular help would be a common Scheme finalization interface (e.g. *wills*) to manage deallocation of C malloc'ed storage.

7. Conclusion

Scheme is a highly adaptable language in which to think concretely about computation. Rather than being cast in concrete, useful software architectures may be cast in jello and easily remolded. This flexibility readily allows a single author to explore a wide range of interesting computational ideas and idioms.

Lao Tze said, "If you are one with the way, the way welcomes you" [17].

After a quarter century of use and dozens of other programming languages, the author finds that Scheme still welcomes him.

Acknowledgments

This space is much too small to list proper acknowledgements to the Scheme implementers, developers, and researchers who have contributed to the author's enjoyment of the language. Those that cannot be elided are the credit to Jeremey Friesen and ioCreative [14] for the background graphic, to the anonymous reviewers, and to Andrew Wright and Bruce Duba for their pattern matching code [30] which made the declarative call interface so much easier to write.

References

[1] **APL** = A Programming Language is described at
http://en.wikipedia.org/wiki/APL_(programming_language)

[2] Jonathan Bachrach jrb@ai.mit.edu
Simple and Efficient Subclass Tests.
http://people.csail.mit.edu/people/jrb/pve/pve.htm
See file OOPS/SRC/SUBCLASS.SCM for details

[3] Darrell Ferguson and Dwight Deugo
Call with Current Continuation Patterns.
In *8th Conference on Pattern Languages of Programs (PLoP 2001)*, 2001.
http://repository.readscheme.org/ftp/papers
/PLoP2001_dferguson0_1.pdf

[4] **Chicken Scheme's FFI** is documented at
http://chicken.wiki.br/Interface to external functions and variables

[5] Craig Chambers and Weimin Chen
Efficient Multiple and Predicate Dispatching.
In *Proceedings of the 1999 ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA '99)*, 1999.
http://citeseer.ist.psu.edu/chambers99efficient.html

[6] **CFFI** is available from and described at
http://common-lisp.net/project/cffi/

[7] Weimin Chen and Volker Turau
Multiple-Dispatching Based on Automata.
In *Journal of Theory and Practice of Object Systems, 1(1)*, 1995.
See file OOPS/SRC/GENERICS.SCM for details.

[8] **Chez Scheme's FFI** is documented at
http://www.scheme.com/csug7/foreign.html#./foreign:h0

[9] **CLOS**, the Common Lisp Object System, is described in
http://en.wikipedia.org/wiki/Common_Lisp_Object _System

[10] The **Dylan** programming language is described in
http://en.wikipedia.org/wiki/Dylan_programming_language

[11] The **Gambit Scheme** system is available at
http://dynamo.iro.umontreal.ca/~gambit/

[12] The **Gambit Scheme's FFI** is documented at
http://www.iro.umontreal.ca/~gambit/doc
/gambit-c.htmlSEC129

[13] The **GUI-Toy** code is available at
http://code.google.com/p/gui-toy/

[14] ioCreative **No Plan B** Graphic is from their web site
http://www.iocreative.com/funstuff

[15] **Ikarus Scheme** is available from
http://www.cs.indiana.edu/~aghuloum/ikarus/

[16] **SDL Worms game** is available from
https://code.launchpad.net/ aghuloum/ikarus-libraries/ikarus-sdl

[17] Lao Tze **Tao Te Ching**

[18] **Larceny Scheme procedure access** is documented in the Larceny User Manual:
http://larceny.ccs.neu.edu/doc/user-manual-alt.html

[19] **Larceny Scheme's FFI** is described at
http://larceny.ccs.neu.edu/doc/LarcenyNotes/note7-ffi.html

[20] Sun's **Lively Kernel** is available from
http://research.sun.com/projects/lively/

[21] **OOPS** is available as part of GUI-Toy for Gambit at
http://dynamo.iro.umontreal.ca/~gambit/wiki
/index.php/Dumping_Grounds

[22] David L. Parnas
On the Criteria to be Used in Decomposing Systems into Modules.
In *Comm. ACM vol 15, December 1972.*

[23] **PLT's FFI** is documented at
http://download.plt-scheme.org/doc/372/html/foreign/

[24] **PSTk** is available from
http://t3x.org/pstk/

[25] The Revised 6 Report on **Scheme** is available at
http://www.r6rs.org/

[26] **SDL**, the Simple Direct-media Layer, is a cross-platform graphics library available from
http://www.libsdl.org/

[27] The **SK8** multimedia development environment is described in
http://en.wikipedia.org/wiki/SK8

[28] **Squeak** is available from
http://www.squeak.org/

[29] **TinyTalk** is available as part of GUI-Toy for Gambit at
http://dynamo.iro.umontreal.ca/~gambit/wiki
/index.php/Dumping_Grounds
A portable R6RS version is available from
https://launchpad.net/kend/

[30] Andrew K. Wright and Bruce F. Duba
Pattern Matching for Scheme
1995
http://en.scientificcommons.org/66521

[31] Yoav Zibin and Joseph Gil
Fast Algorithm for Creating Space Efficient Dispatching Tables with Application to Multi-Dispatching.
OOPSLA '92, 1992
http://citeseer.ist.psu.edu/zibin02fast.html
See file OOPS/SRC/GENERICS.SCM for details.