Programming Languages as Operating Systems (*or* Revenge of the Son of the Lisp Machine)

Matthew Flatt   Robert Bruce Findler   Shriram Krishnamurthi   Matthias Felleisen
Department of Computer Science*
Rice University
Houston, Texas 77005-1892

Abstract

The MrEd virtual machine serves both as the implementation platform for the DrScheme programming environment, and as the underlying Scheme engine for executing expressions and programs entered into DrScheme's read-eval-print loop. We describe the key elements of the MrEd virtual machine for building a programming environment, and we step through the implementation of a miniature version of DrScheme in MrEd. More generally, we show how MrEd defines a high-level operating system for graphical programs.

1 MrEd: A Scheme Machine

The DrScheme programming environment [10] provides students and programmers with a user-friendly environment for developing Scheme programs. To make programming accessible and attractive to novices, DrScheme provides a thoroughly graphical environment and runs under several major windowing systems (Windows, MacOS, and Unix/X). More than 60 universities and high schools currently employ DrScheme in their computing curriculum, and new schools adopt DrScheme every semester.

We implemented DrScheme by first building **MrEd** [15], a portable Scheme [23] implementation with a graphical user interface (GUI) toolbox. MrEd serves both as the implementation platform for DrScheme, and as the underlying Scheme engine for executing expressions and programs entered into DrScheme's read-eval-print loop (REPL). This strategy follows a long tradition of meta-circular implementation that is virtually synonymous with Lisp, and generally understood for high-level languages as a whole [20, 25, 28].

Since DrScheme exposes MrEd's language constructs directly to the REPL, DrScheme can easily execute programs that use the full MrEd language, including its GUI toolbox. At the same time, DrScheme must protect its GUI against interference from the programs it executes, and it must be able to halt a program that has gone awry and reclaim the program's resources—even though the program and DrScheme share a single virtual machine.

To address this problem, MrEd provides a small set of new language constructs. These constructs allow a program-running program, such as DrScheme, to run nested programs directly on the MrEd virtual machine without sacrificing control over the nested programs. As a result, DrScheme can execute a copy of DrScheme that is executing its own copy of DrScheme (see Figure 1). The inner and middle DrSchemes cannot interfere with the operation of the outer DrScheme, and the middle DrScheme cannot interfere with the outer DrScheme's control over the inner DrScheme.

In this paper, we describe the key elements of the MrEd virtual machine, and we step through the implementation of a miniature version of DrScheme in MrEd. More generally, we show how MrEd defines a high-level operating system (OS) for graphical programs. As in other high-level OSes, safety and security in MrEd derive from properties of the underlying programming language. Mere safety, however, provides neither the level of protection between applications nor the kind of process control that conventional OSes provide. Such protection and control is crucial for implementing many kinds of programs, including programming environments and scripting engines. By describing how we implemented DrScheme in MrEd, we demonstrate how to obtain key OS facilities through small extensions to a high-level programming language.

The remainder of the paper is organized as follows. Section 2 sketches a miniature DrScheme, called SchemeEsq, and explains in more detail the implementation challenges for creating a graphical programming environment. Section 3 provides a brief overview of MrEd. Section 4 steps through the implementation of SchemeEsq as a MrEd program. Section 5 explains how MrEd functions as a high-level OS. Section 6 discusses some problems for future work, and Section 7 compares MrEd to related systems.

2 SchemeEsq: The Challenge

SchemeEsq, depicted in Figure 2, is a simple programming shell that captures the essential properties of DrScheme as a program-running program. Roughly, SchemeEsq implements a read-eval-print loop (REPL) that consumes expressions and evaluates them:

```
(define (repl)
  (print (eval (read)))
  (repl))
```

*This research was partially supported by a Lodieska Stockbridge Vaughan Fellowship, NSF grants CCR-9619756, CDA-9713032, and CCR-9708957, and a Texas ATP grant.

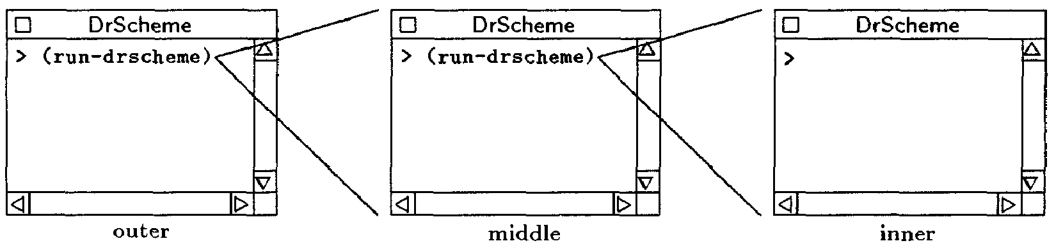
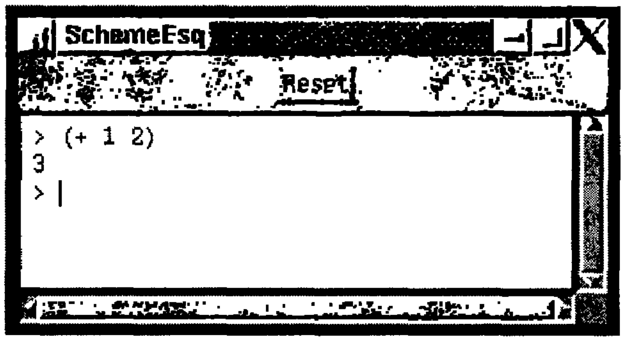


Figure 1: DrScheme in DrScheme in DrScheme



```
> (+ 1 2)
3
> |
```

Figure 2: SchemeEsq

This rough *repl* sketch relies on extensive underlying machinery to implement each of the read, eval, and print steps. SchemeEsq models DrScheme in more detail, showing how read gets characters from the user, where print sends its output, etc. Furthermore, unlike the *repl* function, SchemeEsq demonstrates how to catch errors in the user's program and how to stop a program's execution.

- SchemeEsq's REPL accepts expressions, evaluates them, and displays the results, all within a GUI text editor. For simplicity, we assume that the user submits a complete Scheme expression for evaluation by hitting the Enter key. If a REPL expression signals an error, SchemeEsq prints the error message and creates a new input prompt. At all times, the text preceding the current prompt is locked so the user cannot modify it.
- In addition to standard Scheme, SchemeEsq's REPL provides access to the entire MrEd toolbox, permitting the user's program to create threads and GUI objects. User-created GUI elements must not interfere with SchemeEsq's own GUI. For example, the user's program might create a modal dialog that disables all of the user's other windows, but it must not disable the SchemeEsq window.
- SchemeEsq's **Reset** button stops the current evaluation, reclaiming all resources currently in use by the user's program. All active threads and GUI elements created by the program must be destroyed.
- Although SchemeEsq redirects printed output from the user's program to its REPL window, the program must otherwise execute within SchemeEsq in exactly the same way as it would execute within its own MrEd virtual machine.

The crucial requirement for SchemeEsq is that it must run any program *securely*, in the sense that the program cannot interfere with SchemeEsq's operation.[1] Indeed, since SchemeEsq is itself a MrEd program, SchemeEsq must be able to run copies of itself any number of times and to any nesting depth. No matter how many SchemeEsqs are running and how deeply they are nested, each instance of SchemeEsq must be free from interference from its children. At the same time, a single click to **Reset** in the outermost SchemeEsq must terminate all of the other instances.

3 MrEd Overview

MrEd acts as a high-level OS to service the GUI, security, and scalability needs of SchemeEsq. In the same way that a current production OS (e.g., Unix/X, Microsoft Windows) defines a GUI API, a process model, and a library-linking mechanism, MrEd defines constructs within Scheme for creating GUIs, securely executing untrusted code, and linking together code modules:

- GUI construction — MrEd provides GUI elements via built-in classes that a programmer can instantiate or extend. Event dispatching is automatic and normally synchronous, but MrEd also permits controlled, asynchronous (i.e., parallel) event handling. In addition, MrEd provides special support for multimedia editor applications (hence the *Ed* in *MrEd*), such as word processors and drawing programs.
- Embedding and security — MrEd provides multiple threads of execution, thread-specific configuration, and resource control. These constructs support the secure embedding of programs within other programs that control the embedded execution environment and resource consumption.
- Modularity and extensibility — MrEd's module and class constructs enable the construction of reusable components. These component constructs naturally complement MrEd's object-oriented GUI toolbox, allowing a programmer to implement reusable and composable extensions of GUI elements.

The following sections provide an overview of MrEd's constructs. Section 3.1 describes a few details of MrEd's GUI toolbox, which we provide for the sake of defining a concrete implementation of SchemeEsq. Section 3.2 describes MrEd's

[1] Except by running out of memory; see Section 6.

constructs for program embedding and security. The ideas underlying these constructs form the main contribution of the paper. Finally, Section 3.3 describes MrEd's support for modular and object-oriented programming, which is integral to our model of programs and processes.

Recommendation: Skip the *Experience and Rationale* boxes for a first reading.

3.1 GUI Construction

MrEd provides the basic building blocks of GUI programs—such as frames (top-level windows), modal dialogs, menus, buttons, check boxes, and text fields—via built-in classes that the programmer can instantiate or extend. For example, a programmer creates a frame by instantiating the built-in frame% class:[2]

```
(define frame
  (make-object frame% "Example" #f 400 200))
```

MrEd's make-object procedure takes a class and returns an instance of the class. Extra arguments to make-object serve as initialization arguments for the object, similar to arguments provided with new in Java. For the frame% class, the initialization arguments specify the frame's title, its parent window (#f if none), and its initial size. The above frame is titled **Example**, has no parent, and is 400 pixels wide and 200 pixels tall.

The built-in classes provide various mechanisms for handling GUI events, which MrEd dispatches automatically. For example, when instantiating the button% class, the programmer supplies an event callback procedure to be invoked when the user clicks the button. The following example creates a **Close** button that hides the frame when the user clicks the button:

```
(make-object button% "Close" frame
                     (lambda (button event)
                       (send frame show #f)))
```

The button's callback procedure uses MrEd's **send** form, which calls a method given an object, the method's name, and method arguments. A frame's *show* method takes one argument, a Boolean value that indicates whether to show or hide the frame.

If a window receives multiple kinds of events, MrEd dispatches events to methods of the window instead of to a callback procedure. For example, a drawing canvas receives update events, mouse events, keyboard events, and sizing events; to handle them, a programmer must derive a new class from the built-in canvas% class and override the event-handling methods:

```
(define my-canvas%
  (class canvas% ; my-canvas% extends canvas%
    (override
      (on-char (lambda (event) (display "keyboard")))
      (on-scroll (lambda (event) (display "scroll"))))
    ...))
```

Callbacks, *Experience and Rationale:* For simple controls, such as buttons, the control's action is normally instance-specific, so the action is best specified as a callback in the make-object expression. For more complex GUI elements, such as canvases, event-handling is often common to a class of instances, so method overriding provides a more extensible mechanism for handling events.

[2]By convention, class names end with a percent sign (%) in MrEd. The source code in this paper runs in MrEd version 100.

MrEd's GUI classes also handle the graphical layout of windows. Our example frame demonstrates a simple layout; the frame's elements are lined up top-to-bottom. In general, a programmer specifies the layout of a window by assigning each GUI element to a parent *container*. A vertical container, such as a frame, arranges its children in a column, and a horizontal container arranges its children in a row. A container can be a child of another container; for example, to place two buttons side-by-side in a frame, a programmer creates a horizontal panel for the buttons:

```
(define panel (make-object horizontal-panel% frame))
(make-object button% "Left" panel ...)
(make-object button% "Right" panel ...)
```

A programmer can adjust the minimum width, minimum height, horizontal stretchability, and vertical stretchability of each GUI element. Using these settings, MrEd picks an initial size for each frame, and it repositions controls when the user resizes a frame.

Containers, *Experience and Rationale:* Existing GUI toolboxes provide a variety of mechanisms for geometry management, but our simple container model is intuitive and surprisingly powerful. Although MrEd permits the definition of new containers with arbitrary layout strategies, we implemented DrScheme using only vertical and horizontal containers.

In addition to the basic GUI building blocks, MrEd provides a collection of classes that support a broad spectrum of editor programs, from word processors to drawing programs. The editor framework addresses a wide range of real-world issues for an editor—including cut-and-paste, extensible file formats, and layered text styles—while supporting a high level of extensibility through the class system.

Editors, *Experience and Rationale:* MrEd's editor toolbox provides a foundation for two common kinds of applications:

1. programs that include a sophisticated text editor: MrEd's simple text field control is inadequate for text-intensive applications. Many programs need editors that can handle multiple fonts and non-text items.
2. programs that include a canvas with dragable objects: MrEd's drawing toolbox provides a generic drawing surface for plotting lines and boxes, but many applications need an interactive canvas, where the user can drag and resize individual objects.

The power and flexibility of the editor toolbox make it fairly complex, and using the toolbox requires a solid understanding of its structure and terminology. Nevertheless, enough applications fit one (or both) of the descriptions above to justify the depth and complexity of the toolbox and the learning investment required to use it.

3.2 Embedding and Security

Conventional operating systems support multiple programs through a *process* abstraction that gives each program its own control flow, I/O environment, and resource controls. A process is distinguished primarily by its address space, where separate address spaces serve both as a protection barrier between programs and as a mechanism for defining a program's environment; e.g., the stdout global variable in a Unix C program contains a process-specific value.

In MrEd, separate address spaces are unnecessary for protection between programs, due to the safety properties of the programming language. Nevertheless, separate programs require separate control flow, I/O environments, and resource controls. Instead of providing an all-encompassing process abstraction, MrEd provides specific mechanisms for creating threads of control, dealing with graphical I/O, and managing resources.

3.2.1 Threads and Parameters

MrEd's thread primitive consumes a procedure of no arguments and invokes it in a new thread. The following example spawns a thread that prints "tick" every second:

```
(define (tick-loop)
  (sleep 1) (display "tick") (tick-loop))
(thread tick-loop)
```

Each thread maintains its own collection of system settings, such as the current directory and the current output port. These settings are called *parameters*.[3] A parameter is queried and modified via a *parameter procedure*, such as current-directory or current-output-port. For example, (current-directory) returns the path of the current directory, while (current-directory *dir*) sets the current directory to *dir*.

Modifying a parameter changes its value in the current thread only. Therefore, by setting the current-output-port in the *tick-loop* thread, we can redirect the "tick" printouts without modifying *tick-loop* and without affecting the output of any other thread:

```
(thread (lambda ()
          (current-output-port (open-output-file "ticks"))
          (tick-loop)))
```

A newly-created thread inherits the parameter values of the creating thread. Thus, if *tick-loop* creates its own threads, they also produce output to the "ticks" file.

Parameter inheritance provides an alternative mechanism for setting the output port in the "ticking" thread. Instead of explicitly setting the port within the ticking thread, we could temporarily set the port in the main thread while creating the ticking thread:

```
(parameterize ((current-output-port
                (open-output-file "ticks")))
  (thread tick-loop))
```

A **parameterize** expression sets the value of a parameter during the dynamic extent of its body. In the above example, **parameterize** restores the output port for the main thread after the ticking thread is created, but the ticking thread inherits "ticks" as its current output port.

Since the output port is set before *tick-loop* is called, the ticking thread has no way to access the original output port. In this way, parameters permit securely configuring the environment of a nested program (or any untrusted thread).

Parameters, *Experience and Rationale:* An early version of MrEd supported bundles of parameter values as first-class objects, called *parameterizations*. Two threads could share a parameterization, in which case modifying a parameter in one thread would change the value in both threads.

This generalization turns out to be nearly useless in practice, since shared state is readily available through a parameter whose value is a mutable object. Worse, parameterizations defeat the essential purpose of parameters for separating global state from thread-specific state. With parameterizations, a library routine cannot, for example, freely adjust the current output port, because even a temporary change might affect evaluation in another thread.

[3]The term *parameter*, the parameter procedure convention, and the **parameterize** form in MrEd imitate those of Chez Scheme [9], although Chez does not provide threads.

3.2.2 Eventspaces

An *eventspace* in MrEd is a context for processing GUI events in a sequential manner. Each eventspace maintains its own queue of events, its own collection of frames, and its own *handler thread*. MrEd dispatches events within an eventspace synchronously in the handler thread, while dispatching events from different eventspaces asynchronously in separate handler threads.

Creating an eventspace starts a handler thread for the eventspace implicitly. Only the handler thread dispatches events, but all threads that share an eventspace can queue events, and all threads (regardless of eventspace) can manipulate an accessible GUI object.[4] When a thread creates a top-level window, it assigns the window to the current eventspace as determined by the current-eventspace parameter.

Eventspaces, *Experience and Rationale:* Windows and BeOS also integrate threads with GUI objects, but in fundamentally different ways:

- Windows associates an event queue with every thread, and a thread can manipulate only those windows within its own queue. A programmer can explicitly merge the queues of two threads so that they share an "eventspace," but the queues are merged permanently, so there is no way to change the "eventspace" of a thread.
- BeOS creates a separate handler thread for every top-level window. Programmers must explicitly implement synchronization among top-level windows, but monitors protect many operations on windows.

Eventspaces are more flexible than either of these designs. Compared to Windows, eventspaces more easily accommodate multiple threads that operate on a single set of graphical objects. Compared to BeOS, eventspaces more easily accommodate single-threaded programs with multiple windows. In principle, MrEd's lack of automatic synchronization on objects increases the potential for race conditions, but such race conditions have occurred rarely in practice. While threads sometimes manipulate GUI objects concurrently, they typically call thread-safe primitive methods.

For example, to call a *graphical-tick-loop* procedure that creates a ticking GUI, we parameterize the ticking thread with a new eventspace:[5]

```
(parameterize ((current-eventspace (make-eventspace)))
  (thread graphical-tick-loop))

(define (graphical-tick-loop)
  (letrec ([frame (make-object frame% "Tick")]
           [msg (make-object message% "tick" frame)]
           [loop (lambda (now next)
                   (sleep/yield 1)
                   (send msg set-label now)
                   (loop next now))])
    (send frame show #t)
    (loop "tock" "tick")))
```

The first expression above creates two threads: the plain thread explicitly created by thread, and the handler thread implicitly created by make-eventspace. Instead of creating the plain thread, we can use queue-callback to call *graphical-tick-loop* within the handler thread:

```
(parameterize ((current-eventspace (make-eventspace)))
  (queue-callback graphical-tick-loop))
```

[4]Unlike Java, MrEd provides no automatic synchronization for the methods of a GUI object. The primitive methods of an object, however, are guaranteed to be thread-safe.

[5]The sleep/yield procedure is like sleep, except that it handles events (such as window-update events) while "sleeping."

The queue-callback primitive queues a procedure to be invoked by the handler thread of the current eventspace. The procedure will be invoked in the same way as an event callback for the eventspace.

Each queued procedure is either a high-priority or low-priority callback, indicated by an optional second argument to queue-callback. When a high-priority callback (the default) and a GUI event are both ready for handling, MrEd invokes the high-priority callback. In contrast, when a low-priority callback and a GUI event are both ready for handling, MrEd invokes the GUI event handler. A programmer can use prioritized callbacks to assign priorities to graphical operations, such as low-priority screen refreshing.

3.2.3 Custodians

In the same way that threads generalize per-process concurrency and eventspaces generalize per-process event sequencing, *custodians* generalize per-process resource control.[6] MrEd places every newly-created thread, eventspace, file port, or network connection into the management of the current custodian (as determined by the current-custodian parameter). A program with access to the custodian can terminate all of the custodian's threads and eventspaces and close all of the custodian's ports and network connections. The custodian-shutdown-all procedure issues such a shut-down command to a custodian, immediately reclaiming the resources consumed by the terminated and closed objects.

Using a custodian, we can start *graphical-tick-loop* and permit it to run for only a certain duration, say 200 seconds, before terminating the thread and reclaiming its graphical resources:

```
(define cust (make-custodian))
(parameterize ((current-custodian cust))
  (parameterize ((current-eventspace (make-eventspace)))
    (queue-callback graphical-tick-loop)))
(sleep 200)
(custodian-shutdown-all cust)
```

Although *graphical-tick-loop* could create new custodians, custodians exist within a strict hierarchy. Every new custodian is created as a sub-custodian of the current custodian, and when a custodian receives a shut-down command, it propagates the shut-down command to its sub-custodians. Thus, a program cannot evade a shut-down command by migrating to a custodian that it creates.

Custodians, *Experience and Rationale:* Custodians manage all objects that are protected from garbage collection by references in the low-level system. For example, an active thread is always accessible via the scheduler's run queue—even if no part of the program refers to the thread—and a visible frame is always accessible via the window manager. Such objects require explicit termination to remove the system's reference and to free the object's resources.

An object that has terminated may continue to occupy a small amount of memory. Custodians rely on garbage collection to reclaim the memory for a terminated object, and a thread in a different custodian might retain a reference to such an object. Each operation on a terminable object must therefore check whether its operand has terminated and signal an error if necessary. For GUI objects in MrEd, primitive methods signal an error when the object has terminated.

[6]Custodians are similar to *resource containers* [4].

3.3 Modularity and Extensibility

Parameters, eventspaces, and custodians provide the necessary infrastructure for defining processes without separate address spaces. The resulting process model permits flexible and efficient communication between programs via procedures, methods, and other language constructs. This flexibility blurs the distinction between programs and libraries. For example, a picture-editing program could work either as a stand-alone application or as a part of a word-processing application. More generally, programmers can replace monolithic programs with flexible *software components* that are combined to define applications.

MrEd supports the definition of *units* [14], which are separately compilable and reusable software components. A unit encapsulates a collection of definitions and expressions that are parameterized over imports, and some of the definitions are exported. A programmer links together a collection of units to create a larger unit or a program. MrEd defines *program* to mean a unit with no imports, similar to the way that conventional OSes with dynamic linking (via DLLs or ELF objects) define a program as a certain form of linkable object.

To permit components that are as reusable as possible, a unit linking graph can contain cycles for defining mutually-recursive procedures across unit boundaries. Furthermore, a unit can contain a class definition where the superclass is imported into the unit, even though the source of the imported class is not known at compile time. In the following example, *NoisyCanvasUnit* defines a *noisy-canvas%* class that is derived from an imported *plain-canvas%* class:

```
(define NoisyCanvasUnit
  (unit (import plain-canvas%)
        (export noisy-canvas%)
    (define noisy-canvas%
      (class plain-canvas%
        ...
        (override
         (on-event (lambda (e)
                     (display "canvas event")
                     (super-on-event e))))
        ...))))
```

Since the actual *plain-canvas%* class is not determined until link time, *NoisyCanvasUnit* effectively defines a *mixin* [6, 16, 26], which is a class extension that is parameterized over its superclass. Using mixins, a programmer can mix and match extensions to produce a class with a set of desired properties. This mode of programming is particularly useful for implementing GUIs, where each mixin encapsulates a small behavioral extension of a GUI element.

Units and Mixins, *Experience and Rationale:* Our work cited above for units and mixins provides a theoretical model of the constructs. In practice, MrEd's implementation of units closely follows the theoretical model, except that units normally import and export bundles of names rather than individual names. In contrast, MrEd's implementation of mixins is less expressive than the model, because the implementation does not handle method name collisions. This difference represents a significant compromise in our implementation of mixins, but MrEd's weaker form is sufficiently powerful for most purposes.

4 Implementing SchemeEsq in MrEd

Equipped with the MrEd constructs defined in the previous section, we can implement the SchemeEsq program described in Section 2. First, we create the SchemeEsq GUI using the MrEd toolbox. Then, we use threads, eventspaces, and custodians to implement secure evaluation for REPL expressions. Finally, we discuss how units and mixins let us extend SchemeEsq to implement the full DrScheme environment.

4.1 SchemeEsq GUI

To implement the SchemeEsq GUI, we first create a frame:

```
(define frame
  (make-object frame% "SchemeEsq" #f 400 200))
```

and make it visible:

```
(send frame show #t)
```

Next, we create the reset button to appear at the top of the frame:

```
(define reset-button
  (make-object button% "Reset" frame
               (lambda (b e) (reset-program))))
```

The callback procedure for the reset button ignores its arguments and calls *reset-program*, which we define later. Finally, we create a display area for the REPL, implemented as an *editor canvas*:

```
(define repl-display-canvas
  (make-object editor-canvas% frame))
```

At this point, our SchemeEsq GUI already resembles Figure 2, but the REPL is not yet active. The actual REPL is implemented as a *text editor* that is displayed by the canvas.[7]

The basic functionality needed in SchemeEsq's REPL—including keyboard event handling, scrolling, and cut and paste operations—resides in MrEd's text% editor class. The *esq-text%* class, defined in the appendix, adapts the text% class to the needs of the REPL by overriding methods to specialize the editor's behavior. For example, when the editor receives an Enter/Return key press, it calls the *evaluate* procedure (which we define later).

In addition to handling input, the *esq-text%* class defines an *output* method for printing output from the user's program into the REPL editor. Since the user's program can create many threads, the *output* method needs a special wrapper to convert multi-threaded *output* calls into single-threaded output. The *queue-output* wrapper performs this conversion by changing a method call into a queued, low-priority GUI event:

```
(define esq-eventspace (current-eventspace))
(define (queue-output proc)
  (parameterize ((current-eventspace esq-eventspace))
    (queue-callback proc #f)))
```

Using the new *esq-text%* class, we create an editor instance and install it into the display canvas:

```
(define repl-editor (make-object esq-text%))
(send repl-display-canvas set-editor repl-editor)
```

The SchemeEsq GUI is now complete, but we have not yet implemented *evaluate* (used in *esq-text%*) and *reset-program* (used by *reset-button*'s callback).

[7]MrEd distinguishes between a display and its editor in the same way that Emacs distinguishes between a window and its buffer.

4.2 SchemeEsq Evaluation

When a user hits the Enter key, SchemeEsq evaluates the expression following the current prompt. SchemeEsq ultimately evaluates this expression by calling the built-in eval procedure. But before letting SchemeEsq call eval, we must ensure that code evaluated in the REPL cannot interfere with SchemeEsq itself, since both SchemeEsq and the user's code execute together in MrEd.

Of course, user code must not gain direct access to the frame or editor of SchemeEsq, since it might call methods of the objects inappropriately. We can hide SchemeEsq's implementation from the user's program by putting it into a module and making all definitions private. For now, we continue to define SchemeEsq through top-level definitions, but the appendix shows the final SchemeEsq program encapsulated in a module.

The remaining problems concern the interaction of control flow in the user's program and in SchemeEsq. Threads with parameters, eventspaces, and custodians provide precisely the mechanisms needed to solve these problems.

4.2.1 Threads in SchemeEsq

An unbounded computation in the user's program must not stall SchemeEsq's GUI. Otherwise, the program would prevent the user from clicking SchemeEsq's reset button. To avoid blocking SchemeEsq on a REPL computation, we evaluate user expressions in a separate thread. The following is a first attempt at defining the *evaluate* procedure for evaluating user expressions:[8]

```
(define (evaluate expr-str)
  (thread
   (lambda ()
     (with-handlers ((exn?
                      (lambda (exn)
                        (display (exn-message exn)))))
       (write (eval (read (open-input-string expr-str)))))
     (newline)
     (send repl-editor new-prompt))))
```

Having created a thread to represent the user process, we must configure the process's environment. For simplicity, we define configuration as redirecting output from the user's program (via display or write) to the REPL editor. To redirect output for the user's program, we set the output port in the evaluation thread:

```
(define (evaluate expr-str)
  (thread
   (lambda ()
     (current-output-port user-output-port) ; ← added
     (with-handlers ((exn?
                      (lambda (exn)
                        (display (exn-message exn)))))
       (write (eval (read (open-input-string expr-str)))))
     (newline)
     (send repl-editor new-prompt))))
```

The above assumes that *user-output-port* port acts as a pipe to the REPL editor. We can define *user-output-port* using

[8]The **with-handlers** form specifies predicate-handler pairs that are active during the evaluation of the **with-handlers** body expression. In *evaluate*, the exn? predicate selects the (**lambda** (*exn*) (display (exn-message *exn*))) handler for all types of exceptions. Thus, *evaluate* catches any exception, prints the error message contained in the exception, and resumes the REPL.

make-output-port, a MrEd procedure that creates a port from arbitrary string-printing and port-closing procedures:

```
(define user-output-port
  (make-output-port
   (lambda (s) (send repl-editor output s))
   (lambda () 'nothing-to-close)))
```

In this use of make-output-port, the string-printing procedure sends the string to the REPL editor, and the port-closing procedure does nothing.

4.2.2 Eventspaces in SchemeEsq

Since the user's program and SchemeEsq execute in separate threads, the user's program and SchemeEsq must handle GUI events in parallel. To this end, SchemeEsq creates a new eventspace for the user's program:

```
(define user-eventspace
  (make-eventspace))
```

To execute user code with *user-eventspace*, we might revise *evaluate* to install the eventspace in the same way that we installed *user-output-port*:

```
(define (evaluate expr-str)
  ...
  (thread
   (lambda ()
     (current-eventspace user-eventspace)
     ...)))
```

Alternatively, we could eliminate the call to thread and evaluate expressions in the handler thread of *user-eventspace*. The handler thread is a more appropriate choice, because code that creates and manipulates GUI objects should run in the event-handling thread to avoid race conditions. To evaluate expressions in the handler thread, we treat the evaluation of REPL expressions as a kind of event, queuing evaluation with queue-callback:

```
(define (evaluate expr-str)
  (parameterize ((current-eventspace user-eventspace))
    (queue-callback  ; ← changed  ↖ added
     (lambda ()
       (current-output-port user-output-port)
       (with-handlers ((exn?
                        (lambda (exn)
                          (display (exn-message exn)))))
         (write (eval (read (open-input-string expr-str)))))
       (newline)
       (send repl-editor new-prompt)))))
```

4.2.3 Custodians in SchemeEsq

We complete SchemeEsq by implementing the reset button's action with a custodian. We define *user-custodian* and create the user's eventspace under the management of *user-custodian*:

```
(define user-custodian (make-custodian))

(define user-eventspace
  (parameterize ((current-custodian user-custodian))
    (make-eventspace)))
```

To implement the *reset-program* procedure for the reset button, we issue a shut-down command on *user-custodian* and then reset the REPL editor:

```
(define (reset-program)
  (custodian-shutdown-all user-custodian)
  (parameterize ((current-custodian user-custodian))
    (set! user-eventspace (make-eventspace)))
  (send repl-editor reset))
```

Each reset destroys *user-eventspace* (by issuing a shut-down command to *user-custodian*), making the eventspace unusable. Therefore, *reset-program* creates a new eventspace after each reset.

4.3 Modularity and Extensibility in SchemeEsq

The appendix assembles the pieces that we have developed into a complete implementation of SchemeEsq. The most striking aspect of SchemeEsq's implementation—besides the fact that it fits on one page—is that half of the code exists to drive the REPL editor. In the real DrScheme environment, the REPL is considerably more complex, and its implementation occupies a correspondingly large portion of DrScheme's overall code.

In implementing DrScheme, we tamed the complexity of the GUI by making extensive use of units and mixins. For example, the parenthesis-highlighting extension for an editor is implemented as a mixin in its own unit, and the interactive search interface is another mixin in a separate unit. Using units and mixins in this way, the implementation strategy that we have demonstrated for SchemeEsq scales to the more elaborate implementation of DrScheme.

DrScheme also exploits units for embedding program-like components. For example, DrScheme's help system runs either as a stand-alone application or embedded within the DrScheme programming environment. The help-system unit imports a class that defines a basic frame for the help window. In stand-alone mode, the class implements a frame with a generic menu bar, but when the help system is embedded in DrScheme, the imported class implements a menu bar with DrScheme-specific menus.

5 High-Level Operating Systems

The development of SchemeEsq in MrEd demonstrates how a few carefully defined extensions can transform a high-level programming language into a high-level operating system. A high-level OS permits flexible and efficient communication between programs through common language mechanisms, such as procedures and methods. It also guarantees type and memory safety across programs through language mechanisms, eliminating the need for separate process address spaces and data marshaling. This flexibility increases the potential for extensible and interoperating programs.

Mere safety, however, provides neither the level of protection between applications nor the kind of process control that conventional OSes provide. As an example, SchemeEsq illustrates how a graphical programming environment must protect its GUI from interference from a program executing within the environment. Although language-based safety can prevent a program from trampling on the environment's data structures, it cannot prevent a program from starving the environment process or from leaking resources.

MrEd combines the programming flexibility of a high-level OS with the conventional process controls of a conventional OS. As we have shown, three key extensions make this combination possible: threads with parameters, eventspaces, and custodians. Our approach to building a high-level OS on top of Scheme should apply equally well to other languages, such as ML or Java.

6 Problems for Future Work

Although MrEd provides custodians for resource reclamation, our current implementation does not support *a priori* resource limits on custodians (analogous to memory use limits on a process) or constraints that prevent a program from triggering frequent system-wide garbage collections. Custodians and parameters appear to be good abstractions for expressing these limits, but our memory management technology must be improved to implement them.

Our SchemeEsq example fails to illustrate certain kinds of protection problems, because the communication between SchemeEsq and a user's program is rather limited. For example, the user's program sends output to SchemeEsq by queueing a GUI event. Since the built-in queueing operation is atomic and non-blocking, there is no danger that the user's program will break a communication invariant by killing its own thread. More sophisticated communication protocols require stronger protection during the execution of the protocol. Indeed, merely adding a limit to the size of the output queue in SchemeEsq (so that the user's thread blocks when the queue is full) requires such protection.

One general solution to the protection problem is to create a new thread—owned by SchemeEsq's custodian—for each communication. This techniques solves the problem because thread creation is an atomic operation, and the newly-created thread can execute arbitrarily many instructions without the risk of being killed by the user's program. Unfortunately, thread creation is an expensive operation in MrEd compared to procedure calls, as in many systems. To reduce this cost for common protection idioms, MrEd provides a call-in-nested-thread procedure that creates a child thread, and then blocks the parent thread until the child terminates. By exploiting the mutual exclusion between the parent and child threads, MrEd can eliminate much of the thread-creation and thread-swapping overhead for protection idioms. Using a similar technique, Ford and Lepreau [17] improved the performance of Mach RPC. Nevertheless, a significant overhead remains.

7 Related Work

As a GUI-oriented, high-level language, MrEd shares much in common with Smalltalk [19], Pilot [27], Cedar [32], the Lisp Machine [7], Oberon [33], and JavaOS [31]. All of these systems simplify the implementation of cooperating graphical programs through a high-level language. Although most of these systems support multiple processes, only MrEd provides the kind of process controls that are necessary for implementing a SchemeEsq-like programming environment.[9]

Other related work aims to replicate the safety, security, and resource control of conventional operating systems within a single address space. Architectures such as Alta [2], SPIN [5], J-Kernel [22], and Nemesis [24], emphasize protection within a single address space, but at the expense of program integration through indirect and inefficient calls. For example, the J-Kernel relies on explicit capabilities, and therefore sacrifices the convenience of direct procedure calls and direct data sharing.

Back and Hsieh [1] provide a detailed explanation of the difference between process control and mere safety in a Java-based operating system. They emphasize the importance of the "red line" that separates user code and kernel code in a conventional OS. This red line exists in MrEd, separating low-level built-in primitives from the rest of the system. For example, queue-callback is effectively an atomic operation to the calling thread. MrEd goes one step further, providing programs with the ability to define new layers of red lines. In particular, SchemeEsq defines a red line between itself and the programs that it executes.

Inferno [8] achieves many of the same goals as MrEd, but in a smaller language that is targeted for communications software rather than general-purpose GUI implementation. Balfanz and Gong [3] explore extensions to Java to support multiple processes, particularly multiple processes owned by different users within a single JVM. They derive some of the same constructs that are defined by MrEd, notably eventspaces.

Haggis [11], eXene [18], and Fudgets [21] provide stream-oriented graphical extensions of functional languages. None provides a mechanism for process and resource control, but the functional streams used by these systems makes them less susceptible to cross-process interference than an imperative GUI layer. A combination of stream-oriented GUIs with custodians may be possible.

8 Conclusion

We have shown how key constructs in MrEd—threads with parameters, custodians, and eventspaces—enabled the development of a graphical programming environment. More importantly, the constructs that enabled DrScheme also address problems in the design of a general-purpose, high-level operating system.

Although MrEd was specifically created for DrScheme, MrEd serves as platform for many other applications as well. These applications include a theorem prover [12], a theater lighting system [30], and a mail client, which demonstrate that MrEd's programming model extends to general GUI programming tasks.

Acknowledgements MrEd's GUI toolbox is based on the wxWindows class library [29]. Thanks to Jay Lepreau, Godmar Back, and Pat Tullmann for perspective on existing work in high-level OSes, and thanks to the anonymous reviewers for helpful comments on the original draft of this paper.

References

[1] Back, G. and W. Hsieh. Drawing the red line in Java. In *Proc. IEEE Workshop on Hot Topics in Operating Systems*, March 1999.

[2] Back, G., P. Tullmann, L. Stoller, W. C. Hsieh and J. Lepreau. Java operating systems: Design and implementation. Technical Report UUCS-98-015, University of Utah, 1998.

[3] Balfanz, D. and L. Gong. Experience with secure multi-processing in Java. Technical Report TR-560-97, Princeton University, Computer Science Department, September 1997.

[9]On the Lisp Machine, allowing programmers to tinker with the OS on-the-fly was considered an advantage [7, page 44].

[4] Banga, G., P. Druschel and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. ACM Symposium on Operating System Design and Implementation*, Feburary 1999.

[5] Bershad, B. N., S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proc. ACM Symposium on Operating Systems Principles*, pages 267–284, December 1995.

[6] Bracha, G. and W. Cook. Mixin-based inheritance. In *Proc. Joint ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications and the European Conference on Object-Oriented Programming*, October 1990.

[7] Bromley, H. *Lisp Lore: A Guide to Programming the Lisp Machine*. Kluwer Academic Publishers, 1986.

[8] Dorward, S., R. Pike, D. L. Presotto, D. Ritchie, H. Trickey and P. Winterbottom. Inferno. In *Proc. IEEE Compcon Conference*, pages 241–244, 1997.

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

[10] Findler, R. B., C. Flanagan, M. Flatt, S. Krishnamurthi and M. Felleisen. DrScheme: A pedagogic programming environment for Scheme. In *Proc. International Symposium on Programming Languages: Implementations, Logics, and Programs*, pages 369–388, September 1997.

[11] Finne, S. and S. P. Jones. Composing Haggis. In *Proc. Eurographics Workshop on Programming Paradigms for Computer Graphics*, September 1995.

[12] Fisler, K., S. Krishnamurthi and K. Gray. Implementing extensible theorem provers. Technical Report 99-340, Rice University, 1999.

[13] Flatt, M. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997.

[14] Flatt, M. and M. Felleisen. Units: Cool modules for HOT languages. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 236–248, June 1998.

[15] Flatt, M. and R. B. Findler. PLT MrEd: Graphical toolbox manual. Technical Report TR97-279, Rice University, 1997.

[16] Flatt, M., S. Krishnamurthi and M. Felleisen. Classes and mixins. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 171–183, Janurary 1998.

[17] Ford, B. and J. Lepreau. Evolving Mach 3.0 to a migrating thread model. In *Proc. USENIX Technical Conference and Exhibition*, pages 97–114, Janurary 1994.

[18] Ganser, E. R. and J. H. Reppy. eXene. In *Proc. of the 1991 CMU Workshop on Standard ML*. Carnegie Mellon University, September 1991.

[19] Goldberg, A. and D. Robson. *Smalltalk 80: The Language*. Addison-Wesley, Reading, 1989.

[20] Hagiya, M. Meta-circular interpreter for a strongly typed language. *Journal of Symbolic Computation*, 8(12):651–680, 1989.

[21] Hallgren, T. and M. Carlsson. Programming with fudgets. In Jeuring, J. and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 137–182. Springer, May 1995.

[22] Hawblitzel, C., C.-C. Chang, G. Czajkowski, D. Hu and T. von Eicken. Implementing multiple protection domains in Java. In *Proc. of USENIX Annual Technical Conference*, pages 259–270, June 1998.

[23] Kelsey, R., W. Clinger and J. Rees (Eds.). The revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9), September 1998.

[24] Leslie, I. M., D. McAuley, R. J. Black, T. Roscoe, P. R. Barham, D. M. Evers, R. Fairburns and E. A. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, September 1996.

[25] McCarthy, J. Recursive functions of symbolic expressions and their computation by machine (Part I). *Communications of the ACM*, 3(4):184–195, 1960.

[26] Moon, D. A. Object-oriented programming with Flavors. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–8, November 1986.

[27] Redell, D., Y. Dalal, T. Horsley, H. Lauer, W. Lynch, P. McJones, H. Murray and S. Purcell. Pilot: An operating system for a personal computer. *Communications of the ACM*, 23(2):81–92, Feburary 1980.

[28] Reynolds, J. Definitional interpreters for higher order programming languages. In *ACM Conference Proceedings*, pages 717–740, 1972.

[29] Smart, J. et al. wxWindows. `http://web.ukonline.co.uk/julian.smart/wxwin/`.

[30] Sperber, M. The Lula system for theater lighting control. `http://www-pu.informatik.uni-tuebingen.de/users/sperber/lula/`.

[31] Sun Microsystems, Inc. JavaOS: A standalone Java environment, 1997. `http://www.javasoft.com/products/javaos/javaos.white.html`.

[32] Swinehart, D. C., P. T. Zellweger, R. J. Beach and R. B. Hagmann. A structural view of the Cedar programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):419–490, October 1986.

[33] Wirth, N. and J. Gutknecht. *Project Oberon*. ACM Press, 1992.

Appendix

Syntax Note: The **unit/sig** module form is like **unit**, except that import and export names are bundled together into *signatures*. The built-in **mred^** signature contains all of the classes and procedures provided by MrEd.

```
(define SchemeEsq
 (unit/sig () ;; no exports
  (import mred^)

  ;; The REPL editor class
  (define esq-text%
    (class text% ()
      ;; lexical access to inherited methods:
      (inherit insert last-position get-text erase)
      ;; lexical access to an overridden method:
      (rename (super-on-char on-char))
      ;; private fields:
      (private (prompt-pos 0) (locked? #t))
      (override
        ;; override can-insert? to block pre-prompt inserts:
        (can-insert? (lambda (start len)
                       (and (>= start prompt-pos)
                            (not locked?))))
        ;; override can-delete? to block pre-prompt deletes:
        (can-delete? (lambda (start end)
                       (and (>= start prompt-pos)
                            (not locked?))))
        ;; override on-char to detect Enter/Return:
        (on-char (lambda (c)
                   (super-on-char c)
                   (when (and (eq? (send c get-key-code)
                                   #\return)
                              (not locked?))
                     (set! locked? #t)
                     (evaluate
                      (get-text prompt-pos
                                (last-position)))))))
      (public
        ;; method to insert a new prompt
        (new-prompt (lambda ()
                      (queue-output
                       (lambda ()
                         (set! locked? #f)
                         (insert "> ")
                         (set! prompt-pos (last-position))))))
        ;; method to display output
        (output (lambda (str)
                  (queue-output
                   (lambda ()
                     (let ((was-locked? locked?))
                       (set! locked? #f)
                       (insert str)
                       (set! locked? was-locked?))))))
        ;; method to reset the REPL:
        (reset (lambda ()
                 (set! locked? #f)
                 (set! prompt-pos 0)
                 (erase)
                 (new-prompt))))
      (sequence
        ;; initialize superclass-defined state:
        (super-init)
        ;; create the initial prompt:
        (new-prompt))))

  ;; Queueing REPL output as an event

  (define esq-eventspace (current-eventspace))
  (define (queue-output proc)
    (parameterize ((current-eventspace
                    esq-eventspace))
      (queue-callback proc #f)))

  ;; GUI creation

  (define frame
    (make-object frame% "SchemeEsq" #f 400 200))
  (define reset-button
    (make-object button% "Reset" frame
                 (lambda (b e)
                   (reset-program))))
  (define repl-editor (make-object esq-text%))
  (define repl-display-canvas
    (make-object editor-canvas% frame))
  (send repl-display-canvas set-editor repl-editor)
  (send frame show #t)

  ;; User space initialization

  (define user-custodian (make-custodian))

  (define user-output-port
    (make-output-port
     ;; string printer:
     (lambda (s) (send repl-editor output s))
     ;; closer:
     (lambda () 'nothing-to-close)))

  (define user-eventspace
    (parameterize ((current-custodian user-custodian))
      (make-eventspace)))

  ;; Evaluation and resetting

  (define (evaluate expr-str)
    (parameterize ((current-eventspace user-eventspace))
      (queue-callback
       (lambda ()
         (current-output-port user-output-port)
         (with-handlers ((exn?
                          (lambda (exn)
                            (display
                             (exn-message exn)))))
           (write (eval (read (open-input-string
                               expr-str)))))
         (newline)
         (send repl-editor new-prompt)))))

  (define (reset-program)
    (custodian-shutdown-all user-custodian)
    (parameterize ((current-custodian user-custodian))
      (set! user-eventspace (make-eventspace)))
    (send repl-editor reset))))

;; Run the program module
(invoke-unit/sig SchemeEsq mred^)
```

Syntax Note: The **invoke-unit/sig** form executes the body of a unit given signatures for the unit's imports, and **invoke-unit/sig** satisfies the imports from its lexical and top-level environment.