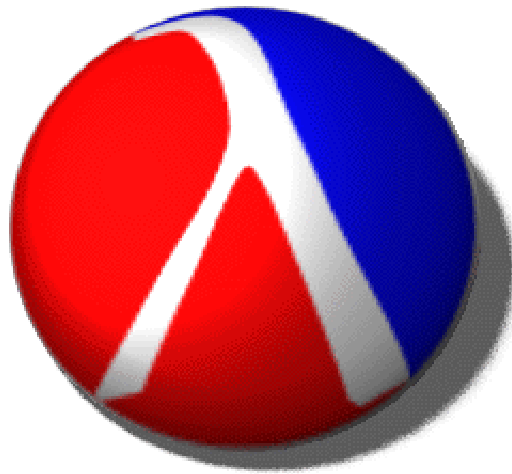
Lightweight Languages as Lightweight Operating Systems



Matthew Flatt

PLT
University of Utah

Premise

programming language

=

operating system

Premise

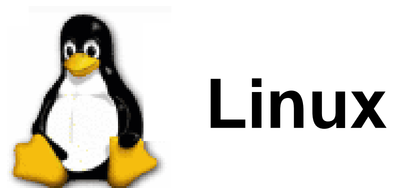
programming language

=

operating system

formal semantics for either is a virtual machine

Operating Systems vs Programming Languages



Operating Systems vs Programming Languages

OS

virtualize machine
for
non-interference

PL

virtualize machine
for expressiveness

Operating Systems vs Programming Languages

OS	**PL**
virtualize machine for non-interference	virtualize machine for expressiveness
$\Rightarrow$ **isolation**	$\Rightarrow$ **cooperation**

Operating Systems vs Programming Languages

OS	PL
virtualize machine for non-interference	virtualize machine for expressiveness
$\Rightarrow$ **isolation**	$\Rightarrow$ **cooperation**

Operating Systems vs Programming Languages

OS ➡ ⬅ PL

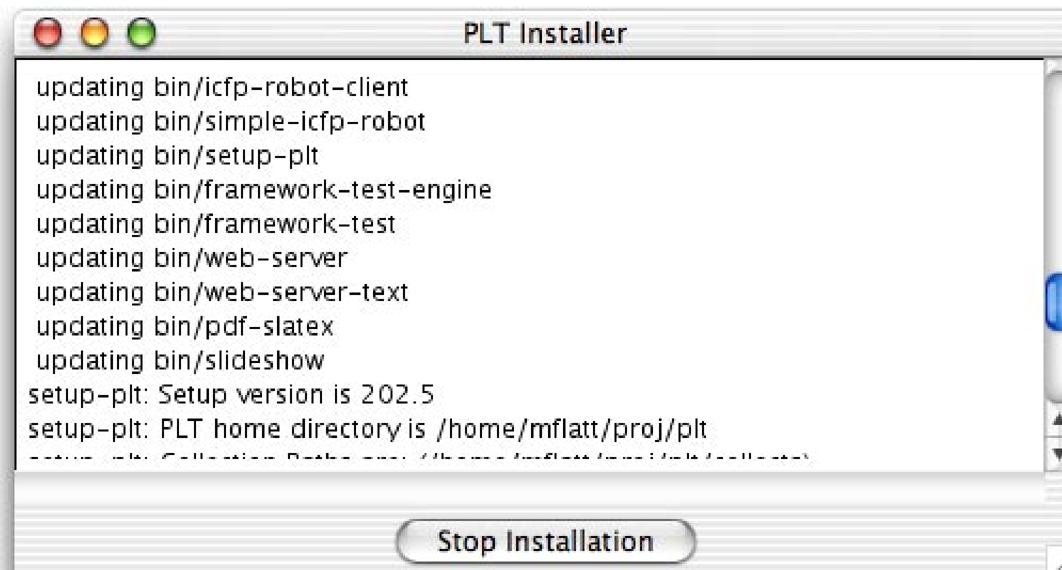
virtualize machine for non-interference

$\Rightarrow$ **isolation**

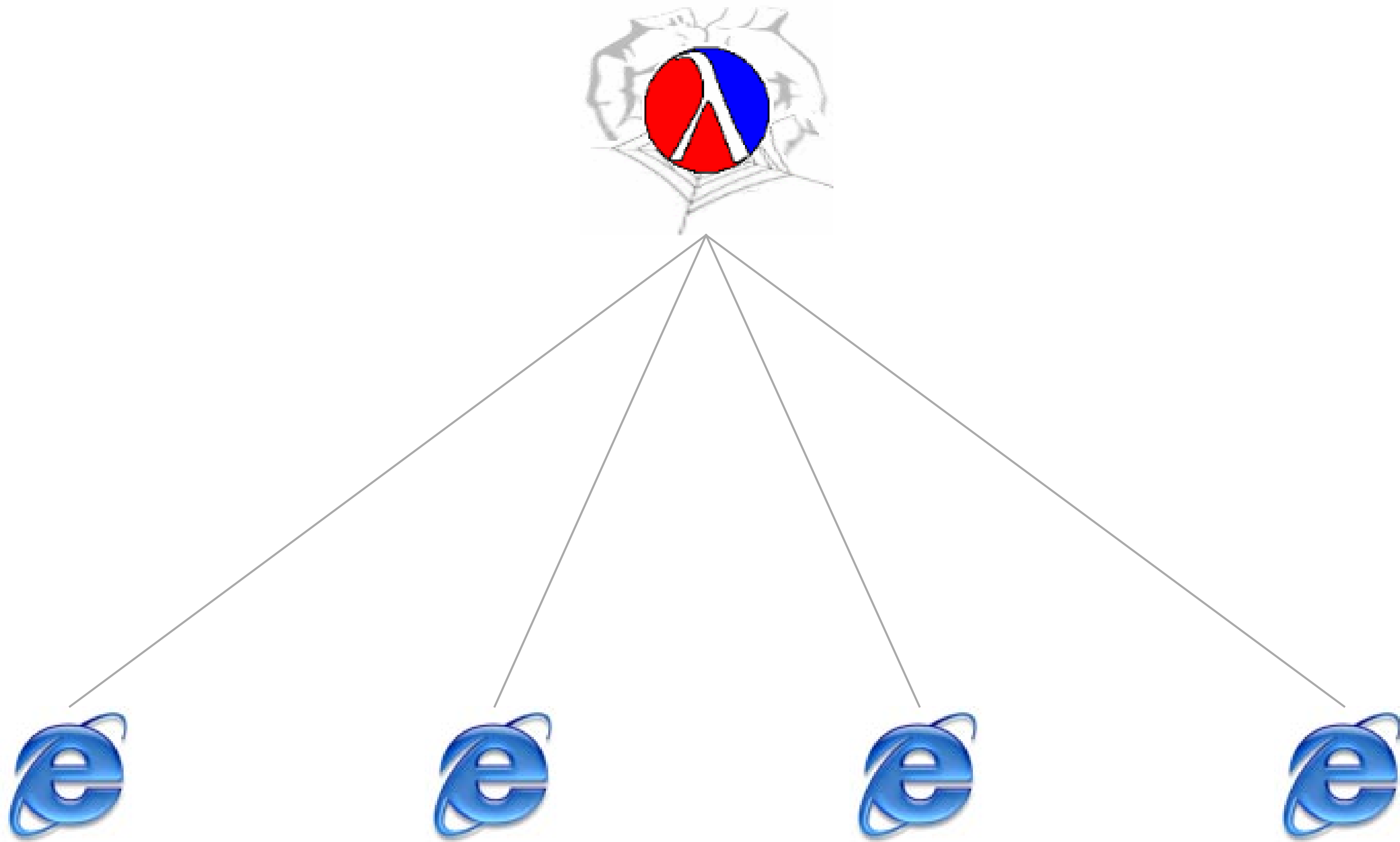
virtualize machine for expressiveness

$\Rightarrow$ **cooperation**

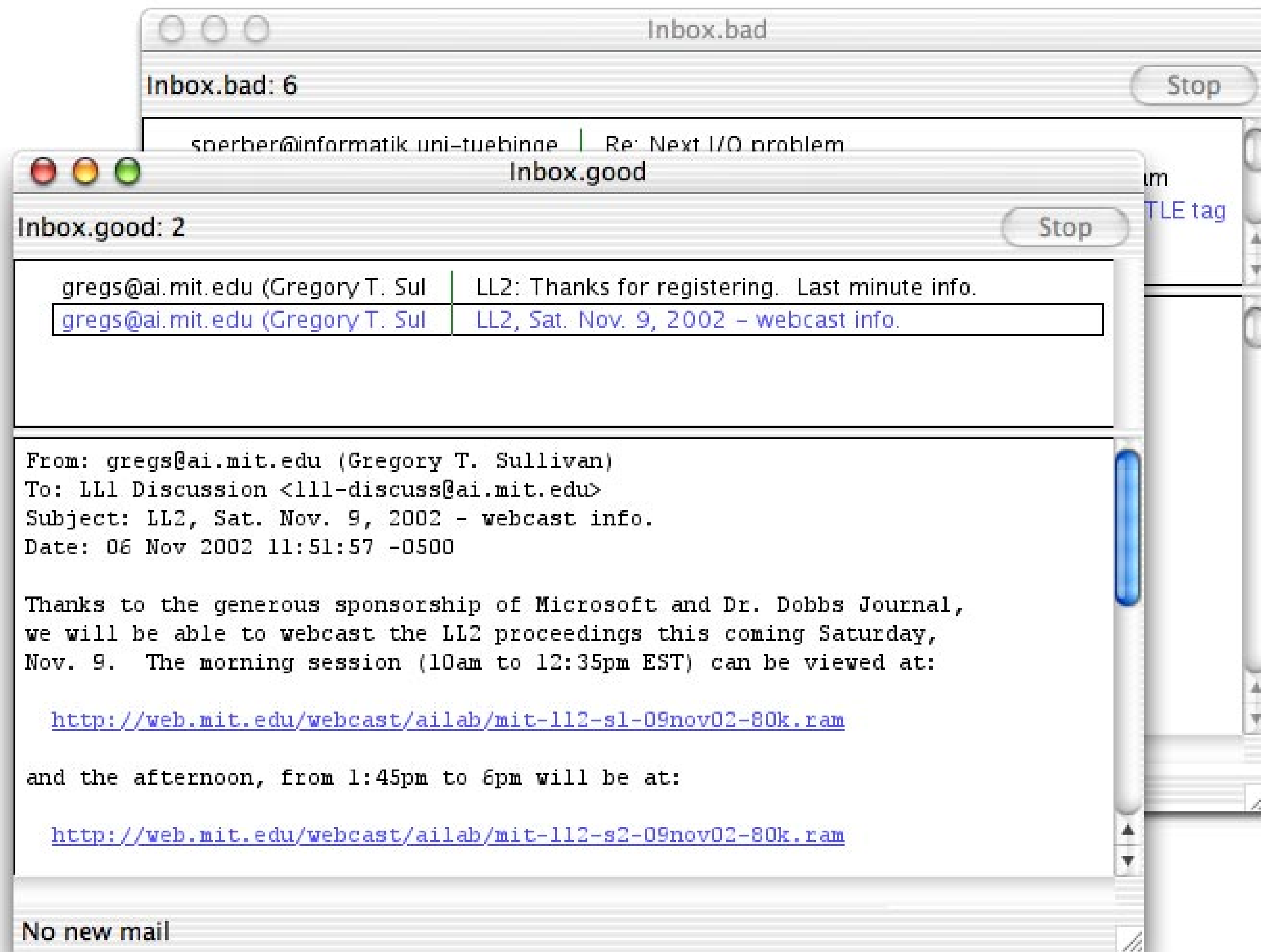
Programming with Processes



Programming with Processes



Programming with Processes



Manifesto

What:

- PL designers should think more as OS designers
- Maintain PL bias toward cooperation

How:

- Break process abstraction into pieces
- Replace isolation with enforced abstraction

Outline

- **Motivation and Manifesto**
- **PLT Scheme as an Example**
  - Threads
  - Parameters
  - Eventspaces
  - Custodians
  - Inspectors
- **Assembling the Pieces**

Threads

Concurrent execution

```
(require "spin-display.scm")
```
eval

```
(define (spin)
  (rotate-a-little)
  (sleep 0.1)
  (spin))

(define spinner (thread spin))
```
eval

```
(kill-thread spinner)
```
eval

Parameters (a.k.a. Fluid Variables)

Thread-local state

```
(printf "Hello\n")
(fprintf (current-output-port) "Hola\n")
(fprintf (current-error-port) "Goodbye\n")
(error "Ciao")
```

eval

```
(parameterize ((current-error-port (current-output-port))
  (error "Au Revoir"))
```

eval

```
(parameterize ((current-error-port (current-output-port))
  (thread
    (lambda ()
      (error "Zai Jian"))))
```

eval

Eventspaces

Concurrent GUIs

```
(thread (lambda () (message-box "One" "Hi")))
(thread (lambda () (message-box "Two" "Bye")))
```

eval

```
(thread (lambda () (message-box "One" "Hi")))
(parameterize ((current-eventspace (make-eventspace)))
  (thread (lambda () (message-box "Two" "Bye"))))
```

eval

Custodians

Termination and clean-up

```
(define c (make-custodian))
(parameterize ((current-custodian c))
  ...)
```

eval

```
(custodian-shutdown-all c)
```

eval

Inspectors

Debugging access to data

```
(define-struct fish (color weight))
(define eddie (make-fish 'red 100))
(print eddie)
```

eval

```
(define senior-inspector (current-inspector))
(parameterize ([current-inspector (make-inspector)])
  (define-struct fish (color weight))
  (define eddie (make-fish 'red 100))
  (parameterize ([current-inspector senior-inspector])
    (print eddie)))
```

eval

Etc.

- Security Guards

Resource access control

- Namespaces

Global bindings

- Will Executors

Timing of finalizations

Outline

- **Motivation and Manifesto**
- **PLT Scheme as an Example**
- **Assembling the Pieces**
  - SchemeEsq, a mini DrScheme [ICFP99]

GUI - Frame

```
(define frame
  (instantiate frame% ()
    [label "SchemeEsq"]
    [width 400] [height 175]))

(send frame show #t)
```

eval

GUI - Reset Button

```
(instantiate button% ()
  [label "Reset"]
  [parent frame]
  [callback (lambda (b e) (reset-program))])
```

eval

GUI - Interaction Area

```
(define repl-display-canvas
  (instantiate editor-canvas% ()
    [parent frame]))
```

eval

GUI - Interaction Buffer

```
(define esq-text%
  (class text% ... (evaluate str) ...))

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

eval

Evaluator

```
(define (evaluate expr-str)
  (thread
    (lambda ()
      (print (eval (read (open-input-string expr-str))))
      (newline)
      (send repl-editor new-prompt))))
```

eval

Evaluator Output

```
(define user-output-port
  (make-custom-output-port ... repl-editor ...))

(define (evaluate expr-str)
  (parameterize ((current-output-port user-output-port))
    (thread
      (lambda ()
        ...))))
```

eval

Evaluating GUIs

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

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

eval

Custodian for Evaluation

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

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

(define (evaluate expr-str)
  (parameterize ((current-output-port user-output-port)
                 (current-eventspace user-eventspace)
                 (current-custodian user-custodian))
    (thread
      (lambda ()
        ...))))
```

eval

Reset Evaluation

```
(define (reset-program)
  (custodian-shutdown-all user-custodian)

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

eval

Inspecting Results

```
(define esq-inspector (current-inspector))
(define user-inspector (make-inspector))

(define (evaluate expr-str)
  (parameterize (...
                 (current-inspector user-inspector))
    (thread
      (lambda ()
        (let ([v (eval ...)])
          (parameterize ((current-inspector esq-inspector))
            (print v))
        ...))))
```

eval

Conclusion

- Programmers need OS-like constructs in languages
  - run-time environment (e.g., script GUI wrapper, talk demos)
  - easy termination (e.g., server sessions, awry demos)
  - concurrency (e.g., browser cancel, mail client)
- Multiple language constructs for "process"
  - programmer can mix and match to balance isolation and cooperation