Fun for Freshman Kids

Felleisen, Findler, Flatt, Krishnamurthi
PLT
Question: Should functional programming be the first thing [programmers] learn?
Question: Should functional programming be the first thing [programmers] learn?

Answer: I don't actually have a very strong opinion on that. I think there are a lot of related factors, such as what the students will put up with! I think student motivation is very important, so teaching students a language they have heard of as their first language has a powerful motivational factor.
game by inner city middle school student, Feb. '09
Christine’s programming language is *pure middle school mathematics* (in Scheme syntax of course):

variable expressions, functions, conditional functions, function composition.
Christine’s programming language is pure middle school mathematics (in Scheme syntax of course):

variable expressions, functions, conditional functions, function composition.

Trick: Christine doesn’t know that it’s mathematics.
<table>
<thead>
<tr>
<th>middle school</th>
<th>highschool</th>
<th>college freshman I</th>
<th>freshman II</th>
</tr>
</thead>
<tbody>
<tr>
<td>expressions, functions, composition</td>
<td>structs, lists, sequences</td>
<td>trees, graphs, higher-order accumulators, gen. rec., &amp; conc. distr. programming</td>
<td>theorems about code, OOP (imperative)</td>
</tr>
</tbody>
</table>
middle school  |  highschool  |  college freshman I  |  freshman II

mathematics  |  programming  |  design & model  |  programming & computing

expressions, functions, composition  |  structs, lists, sequences  |  trees, graphs, higher-order accumulators, gen. rec., &&& conc. distr. programming  |  theorems about code, OOP (imperative)
middle school  highschool  college freshman I  freshman II

mathematics  programming

Emmanuel Schanzer (Harvard)

expressions, functions, composition

structures, lists, sequences

design & model  programming & computing

trees, graphs  higher-order

gener. rec.  accumulators

OOP (imperative)

&&&

conc. distr.  programming

Core PLT Team
How did we get there?
A Functional I/O System

Felleisen, Findler, Flatt, Krishnamurthi
PLT
**Diagnosis:** Students wish to write programs like those that they use, with interactive GUIs.

Apparently functional programming languages must abandon “purity” via monads and/or other advanced type machinery to compete with imperative languages.

Manuel Chakravarty and Gabriele Keller, J. Functional Programming, volume 14(1)
The Abstract Idea:

Turn mathematical functions into event handlers. The underlying OS performs all imperative actions.
The Abstract Idea:

Turn mathematical functions into event handlers. The underlying OS performs all imperative actions.

The Concrete Idea:

Think of the world as a collection of states, clock ticks, mouse events, ... trigger transitions.
The Abstract Idea:

Turn mathematical functions into event handlers. The underlying OS performs all imperative actions.

The Concrete Idea:

Think of the world as a collection of states, clock ticks, mouse events, ... trigger transitions.

The Best Part:

No threading required. No monad. No arrows. No nothing.
import
type World

export
val big-bang : World
  * (World -> World)
  * (World KeyEvt -> World)
  * (World Nat Nat MouseEvt -> World)
  -> World

The OS
w1
w2
w3
w4
import
type World

export
val big-bang : World
  * (World -> World)
  * (World KeyEvt -> World)
  * (World Nat Nat MouseEvt -> World)
  -> World

The OS
import
type World

export
val big-bang : World
  * (World -> World)
  * (World KeyEvt -> World)
  * (World Nat Nat MouseEvt -> World)
  -> World
import

type World

export

val big-bang : World
  * (World -> World)
  * (World KeyEvt -> World)
  * (World Nat Nat MouseEvt -> World)
  -> World
import
type World

export
val big-bang : World
  * (World -> World)
  * (World KeyEvt -> World)
  * (World Nat Nat MouseEvt -> World)
  -> World
import
type World

export
val big-bang : World
    * (World -> World)
    * (World KeyEvt -> World)
    * (World Nat Nat MouseEvt -> World)
    -> World

// call for every clock tick
import
type World

export
val big-bang : World
    * (World -> World)
    * (World KeyEvt -> World)
    * (World Nat Nat MouseEvt -> World)
    -> World

type KeyEvt <= String
val key=? : KeyEvt KeyEvt -> Boolean
import
type World

export
val big-bang : World
  * (World -> World)
  * (World KeyEvt -> World)
  * (World Nat Nat MouseEvt -> World)
-> World

type KeyEvt <= String
val key=? : KeyEvt KeyEvt -> Boolean

type MouseEvt = “button-down” | ...
val mouse=? : MouseEvt MouseEvt -> Boolean
import
type World

export
val big-bang : World
  * (World -> World)
  * (World KeyEvt -> World)
  * (World Nat Nat MouseEvt -> World)
-> World

the final world

type KeyEvt <= String
val key=? : KeyEvt KeyEvt -> Boolean

type MouseEvt <= String
val Mouse=? : MouseEvt MouseEvt -> Boolean
import
  type World

export
  val big-bang : World
    * (World -> World)
    * (World KeyEvt -> World)
    * (World Nat Nat MouseEvt -> World)
    * (World -> Boolean)
  -> World

  type KeyEvt <= String
  val key=? : KeyEvt KeyEvt -> Boolean

  type MouseEvt <= String
  val Mouse=? : MouseEvt MouseEvt -> Boolean
```ocaml
import
  type World

export
  val big-bang : World
    * (World -> World)
    * (World KeyEvt -> World)
    * (World Nat Nat MouseEvt -> World)
    * (World -> Boolean)
    * (World -> Image) -> World

  type KeyEvt <= String
  val key=? : KeyEvt KeyEvt -> Boolean

  type MouseEvt <= String
  val Mouse=? : MouseEvt MouseEvt -> Boolean
```

render the world
**Universe**

**import**

`type World`

**export**

`val big-bang : World`

\[
* (World -> World)
* (World KeyEvt -> World)
* (World MouseEvt -> World)
* (World -> Boolean)
* (World -> Image)
\]

```
-> World
```

`type KeyEvt <= String`

`val key=? : KeyEvt KeyEvt -> Boolean`

`type MouseEvt <= String`

`val Mouse=? : MouseEvt MouseEvt -> ...

...`
import

```
import
type World
```

```
export

val big-bang : World
  * (World -> World)
  * (World KeyEvt -> World)
  * (World MouseEvt -> World)
  * (World -> Boolean)
  * (World -> Image)
  -> World
```

type KeyEvt <= String
val key=? : KeyEvt KeyEvt -> Boolean

type MouseEvt <= String
val Mouse=? : MouseEvt MouseEvt -> ...

---

Student Program

```
export
type World = ...
```

```
import

big-bang
```

```
local

event handlers for
clock ticks
keyboard events
mouse events
```

```
renderer for translating
states of the world into
images
```

```
a stop? predicate
```
import type World

export
type World

val big-bang : World
* (World -> World)
* (World KeyEvt -> World)
* (World MouseEvt -> World)
* (World -> Boolean)
* (World -> Image)
-> World

(type KeyEvt <= String
val key=? : KeyEvt KeyEvt -> Boolean

(type MouseEvt <= String
val Mouse=? : MouseEvt MouseEvt -> ...)

export type World = ...

import

local
event handlers for
  clock ticks
  keyboard events
  mouse events

renderer for translating
  states of the world into
  images

a stop? predicate

(Quelle et al: Units)
That’s for you.

Reality: In PLT, *big-bang* is just a “little” language (aka macro) for describing worlds.
;;; World = NaturalNumber
;;; interpretation: the distance of the LANDER from top

;;; World -> World
(define (run y0)
    (big-bang y0 (on-draw to-image) (on-tick drop)))

;;; World -> World
(define (drop y) (+ y 3))

;;; World -> Image
(define (to-image y)
    (place-image 400 y MOON))

(define MOON ...)
World = NaturalNumber (0, 1, 2, ...)
;; interpretation: the distance of the LANDER from top

(define (run y0)
  (big-bang y0 (on-draw to-image) (on-tick drop)))

(define (drop y) (+ y 3))

(define (to-image y)
  (place-image 400 y MOON))

(define MOON ...)

Example
;; World = NaturalNumber
;; interpretation: the distance of the LANDER from top

;; World -> World
(define (run y0)
  (big-bang y0 (on-draw to-image) (on-tick drop)))

;; World -> World
(define (drop y) (+ y 3))

;; World -> Image
(define (to-image y)
  (place-image 400 y MOON))

(define MOON ...)

Example
;; World = NaturalNumber
;; interpretation: the distance of the LANDER from top

;; World -> World
(define (run y0)
  (big-bang y0 (on-draw to-image) (on-tick drop)))

;; World -> World
(define (drop y) (+ y 3))

;; World -> Image
(define (to-image y)
  (place-image 400 y MOON))

(define MOON ...)
;; World = NaturalNumber
;; interpretation: the distance of the LANDER from top

;; World -> World
(define (run y0)
  (big-bang y0 (on-draw to-image) (on-tick drop)))

;; World -> World
(define (drop y) (+ y 3))

;; World -> Image
(define (to-image y)
  (place-image ... 400 y MOON))

(define MOON ...)
;; World = NaturalNumber
;; interpretation: the distance of the LANDER from top

;; World -> World
(define (run y0)
  (big-bang y0 (on-draw to-image) (on-tick drop)))

;; World -> World
(define (drop y) (+ y 3))

;; World -> Image
(define (to-image y)
  (place-image 400 y MOON))

(define MOON ... )
;; World = NaturalNumber
;; interpretation: the distance of the LANDER from top

;; World -> World
(define (run y0)
  (big-bang y0 (on-draw to-image) (on-tick drop)))

;; World -> World
(define (drop y) (+ y 3))

;; World -> Image
(define (to-image y)
  (place-image 400 y MOON))

(define MOON ...)

Example
World is one of:
-- "red"
-- "green"
-- "yellow"
interpretation: the current state of the traffic light

World KeyEvt -> World
(define (world-switch s ke)
  (cond
   [(key=? ke " ") (light-switch s)]
   [else s]))

World -> World
(define (light-switch s)
  (cond
   [(string=? "red" s) "green"]
   [(string=? "green" s) "yellow"]
   [(string=? "yellow" s) "red"]))

World -> Image
(define (world-render s)
  (cond
   [(string=? "red" s) (place-image RED X YRED LIGHT)]
   [(string=? "green" s) (place-image GREEN X YGREEN LIGHT)]
   [(string=? "yellow" s) (place-image YELLOW X YYELLOW LIGHT)]))

World -> World
(define (world-run s0)
  (big-bang s0
    (on-draw world-render)
    (on-key world-switch)))

geometric constants
(define RADIUS 20)
World is one of:
  -- "red"
  -- "green"
  -- "yellow"
; interpretation: the current state of the traffic light

World KeyEvt -> World
(define (world-switch s ke)
  (cond
    [(key=? ke " ") (light-switch s)]
    [else s]))

World -> World
(define (light-switch s)
  (cond
    [(string=? "red" s) "green"]
    [(string=? "green" s) "yellow"]
    [(string=? "yellow" s) "red"])))

World -> Image
(define (world-render s)
  (cond
    [(string=? "red" s) (place-image RED X YRED LIGHT)]
    [(string=? "green" s) (place-image GREEN X YGREEN LIGHT)]
    [(string=? "yellow" s) (place-image YELLOW X YYELLOW LIGHT)]))

World -> World
(define (world-run s0)
  (big-bang s0
    (on-draw world-render)
    (on-key world-switch)))

; geometric constants
(define RADIUS 20)
...
The World is Not Enough
Universe Programs for Connecting Distributed Worlds
(define (chat-room n)
  (big-bang world0
    (name n)
    (on-key react) ;; World KeyEvt -> World | World * Message
    (on-draw render)
    (on-receive receive) ;; World * Message -> World * Message
    (register "america.ccs.neu.edu")))
(define (chat-room n)
  (big-bang world0
    (name n)
    (on-key react) ;; World KeyEvt -> World | World * Message
    (on-draw render)
    (on-receive receive) ;; World * Message -> World * Message
    (register "america.ccs.neu.edu")))

(define (run _)
  (universe '()
    (on-new new-chatter)
    (on-msg forward)))
• the server can “pass through” packages (chat)
• the server can play referee (distributed games)
• the server can fake peer-to-peer (“Napster”)
• ... and it is all **functional**
• examples: binary games, distributed games, chat rooms, maze explorations;
Squint once and you get theorems ...
World = NaturalNumber
;; interp.: the distance of the LANDER from top

World -> World
(defun run (y0)
  (big-bang y0 (on-draw to-image) (on-tick drop)
    (stop-when? too-low-on-screen)))

World -> World
(defun drop (y) (+ y 1))

World -> Image
(defun to-image (y)
  (place-image 400 y MOON))

(defthm lander-within-picture ...)

theorem proving for freshmen
(defun run (y0)
  (big-bang y0 (on-draw to-image) (on-tick drop)
    (stop-when? too-low-on-screen)))

(defun drop (y) (+ y 1))

(defun to-image (y)
  (place-image 400 y MOON))

(defthm lander-within-picture ...)
Squint twice and you get objects ...
(define-struct ufo (x y dx dy))
;; UFO = (make-ufo Nat Nat Integer Integer)
;; interp. the UFO’s current location and velocity

;; UFO -> UFO
;; move this UFO for one tick
(define (ufo-move/tick u)
  (make-ufo (+ (ufo-x u) (ufo-dx u)) (ufo-y u) (ufo-dx u) (ufo-dy u))))

;; UFO Image -> Image
;; add this UFO to the given scene
(define (ufo-render u s)
  (place-image UFO (ufo-x u) (ufo-y u) s))

...
(define ufo% (class object% (init-field x y dx dy))
 ;; UFO = (new ufo% Nat Nat Integer Integer)
 ;; interp. the UFO’s current location and velocity

 ;; -> UFO
 ;; move this UFO for one tick
(define/public (move/tick)
 (new ufo% (+ x dx) y dx dy))

 ;; UFO Image -> Image
 ;; add this UFO to the given scene
(define/public (render s)
 (place-image UFO x y s))
...))

Applicative Objects
(define ufo
  (class object
    (init-field x y dx dy))
;; UFO = (make-ufo Nat Nat Integer Integer)
;; interp. the UFO’s current location and velocity

;; -> VOID
;; move this UFO for one tick
(define/public (move/tick)
  (set! x (+ x dx)))

;; UFO Image -> Image
;; add this UFO to the given scene
(define/public (render s)
  (place-image UFO x y s))
...)

plus Kathryn Gray (Cambridge) & Viera Proulx (NEU)
(define ufo
  (class object
    (init-field x y dx dy))
;; UFO = (make-ufo Nat Nat Integer Integer)
;; interp. the UFO’s current location and velocity

;; -> VOID
;; move this UFO for one tick
(define/public (move/tick)
  (set! x (+ x dx)))

;; UFO Image -> Image
;; add this UFO to the given scene
(define/public (render s)
  (place-image UFO x y s))
...))
Some Conclusions
• functional I/O is a key technology for teaching functional programming at all levels, starting with middle school

• it naturally generalizes to a distributed (concurrent) universe of worlds

• ... and provides a natural path to teaching theorem proving about programs early

• ... and enables a motivated and smooth transition to imperative OOP design
Try it out in your languages. It’s easy.
Try it out in your languages. It’s easy.

(2000 lines, when you’re standing on the toes of great co-developers)
The End

Thanks to:
Carl Eastlund (NEU)
Kathi Fisler (WPI)
Emmanuel Schanzer (Harvard U.)