A Functional I/O System *

or, Fun for Freshman Kids

Matthias Felleisen
Northeastern University

Robert Bruce Findler
Northwestern University

Matthew Flatt
University of Utah

Shriram Krishnamurthi
Brown University

Abstract

Functional programming languages ought to play a central role in mathematics education for middle schools (age range: 10–14). After all, functional programming *is* a form of algebra and programming is a creative activity about problem solving. Introducing it into mathematics courses would make pre-algebra course come alive. If input and output were invisible, students could implement fun simulations, animations, and even interactive and distributed games all while using nothing more than plain mathematics.

We have implemented this vision with a simple framework for purely functional I/O. Using this framework, students design, implement, and test plain mathematical functions over numbers, booleans, string, and images. Then the framework wires them up to devices and performs all the translation from external information to internal data (and vice versa)—just like every other operating system. Once middle school students are hooked on this form of programming, our curriculum provides a smooth path for them from pre-algebra to freshman courses in college on object-oriented design and theorem proving.

Categories and Subject Descriptors D.2.10 [*Software Engineering*]: Design—methodologies; D.4.7 [*Operating Systems*]: Organization and Design—interactive systems ***General Terms*** Design, Languages ***Keywords*** Introductory Programming

1. Functions for Freshmen

Based on our decade-long experience (Felleisen et al. 2004a), novices to programming tend to accept languages that they haven't heard of—as long as they can quickly construct a program that is like the applications they use on their computers. To this end, the chosen language must come with a rich framework for input and output (I/O), ideally via graphical interfaces. Chakravarty and Keller (2004) present corroborating evidence based on a thorough analysis of their Haskell-based introductory courses. They also report, however, that most Haskell texts deemphasize I/O. Our own review shows that three (Thompson 1997; Bird and Wadler 1998; Hutton 2007) of four major Haskell-based text books introduce I/O in the last third of the book or the final chapter; only Hudak (2000) tackles it head-on, though in a quasi-imperative manner. Surprisingly, even O'Sullivan et al. (2008)'s *Real World Haskell* has difficulties explaining I/O, according to some on-line reviews.

Here we present our approach to reconciling I/O with purely functional programming, especially for a pedagogical setting. The I/O framework extends the DrScheme (Findler et al. 2002) teaching languages for our text *How to Design Programs* (HtDP) (Felleisen et al. 2001), but it is also accessible from other dialects. Our framework does not require the use of any monads or other threading devices, meaning middle school students can write animation and interactive games as a bunch of mathematical functions. Indeed, because everything is just a function on numbers, strings, and images, students can also test every step as they design their programs. For the past three years, the curriculum has been deployed at eight middle schools (ages 10–14). The students tend to embrace programming enthusiastically after a nine-week course. Moreover, this kind of programming experience seems to improve the performance of these students in standard mathematics courses.

The purpose of our paper is to share our technical development so that others can duplicate our pedagogical experiences, whether the functional I/O library is layered on top of an imperative library (as in our implementation) or on top of an I/O monad (as an implementation for Haskell would be). The next section provides an overview of our experiences to explain some of the sociological context where we apply functional programming with I/O. Section 3 illustrates how we start middle school students on functional programming. The rest of the paper focuses on the technical foundations.Sections 4 and 5 explain the I/O library and how to use it. Section 6 is about the college-level curricular context. In section 7, we compare our approach to the Clean Event I/O system, which is closest to our framework, and to some other pieces of work.

2. Experience

Age group	Since	Framework	Schools
middle school	2006	animation	eight
two- or three-object games, simplistic sound			
high school	2004	animation	≥ 30
N object games, simulations			
college	2003	animation	≥ 20
N object games, simulations, visualization			
college & MS program	2008	dist. prog.	Chicago & NEU
distributed two-player games			

Variants of our I/O library have been in use since 2003 at a variety of places. The nearby table provides a concise overview of the kinds of students, sites, and projects that it enables. Clearly, the library is primarily useful in college-level freshman courses. Starting in 2004, we also introduced the library into the TeachScheme! workshops; some 30 schools have run courses with it. Three years ago, Emmanuel Schanzer derived the Bootstrap curriculum (`www.bootstrapworld.org`) from HtDP for Citizen Schools (`www.citizenschools.org`), which runs after-school programs

* This research was partially supported by several grants from the National Science Foundation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP'09, August 31–September 2, 2009, Edinburgh, Scotland, UK.
Copyright © 2009 ACM 978-1-60558-332-7/09/08…$5.00

on a wide range of topics in poor neighborhoods across the US. At this point, eight sites (in Austin, the Bay Area, Boston and surroundings, and New York City) have used Bootstrap with a specialized variant of the I/O library. Finally, the first author uses the library in an immersion course for Northeastern's MS program.

Middle school students in the Bootstrap after-school courses typically have little background in mathematics. They are barely comfortable with calculations on numbers; they struggle with variables, if they have encountered them at all; and many are discouraged about the entire topic. Occasionally some of the students are concurrently enrolled in a first pre-algebra course.

One goal of Bootstrap is to bring across the notion of a function as something that relates quantities, though not necessarily numbers. Naturally, the citizen teachers (volunteer instructors working for Citizen Schools) do not tell students that they are about to study a parenthesized form of algebra. Instead, they demonstrate interactive computer games and encourage students to think about creating such games on their own.

A normal Citizen Schools course lasts nine weeks, with one two-hour session per week and no homework assignments. Within this time, most students design and systematically implement an interactive game of their choice that involves a fixed number of moving objects, object collisions, and score keeping. They code these games in the "Beginning Student" language of HtDP, extended with our new library. Citizen teachers report that the majority of students actively and enthusiastically participate in these courses, many asking for more when the course ends.[1,2]

By the end of the course, some of the citizen teachers share with the students that their game programs *are* mathematics. Anecdotal evidence suggests that making mathematics come alive in this manner has a direct impact on students' performance in subsequent mathematics courses. Numerous instructors report conversations with mathematics teachers on how the students have changed their attitude about mathematics and how their grades have improved.

The I/O library plays a similarly important role in high school courses and first-year college curricula as it does for Bootstrap. Many high school teachers and college instructors have noticed that students fail to understand mathematical functions. An introductory curriculum that makes functions critical and fun for animations, simulations, and interactive and multi-player/multi-computer games can thus play a central role in enhancing students' preparation. Once students understand the basic idea of a function, it is easy to motivate them to study the systematic design of functions as advertised in HtDP (Felleisen et al. 2004b). After all, designing properly enables them to write more interesting simulations, animations, games, etc. As we explain in section 6, this kind of first course also prepares students well for the rest of the first year, including applicative and imperative object-oriented programming.

Some typical examples are the freshman courses at the University of Chicago and Northeastern University. Although Chicago uses a quarter system, its course reaches the same milestones as Northeastern's, due to the small class size at Chicago and students' strong academic preparation. In both courses, students work out at least two complete game project in a purely functional setting, often the same ones at both places (2008: Snake, Chat Noir). The first project (between 500 and 1,000 lines) is repeated twice, once to ensure students can implement suggestions and a second time to introduce abstraction via higher-order functions. The second project (some 1,000–2,000 loc) is typically an end-of-semester/quarter project where students have some creative freedom, too. The authors have repeatedly observed that students continue to work on their game programs after the semester/quarter ends.

3. Arithmetic, Algebra, and Movies

In middle school, mathematics teachers often ask students to determine the next number in a sequence such as $1, 4, 9$. Or they show students a series of shapes and ask what the next shape should look like. Eventually these series are arranged in the form of tables, e.g.,

$x =$	0	1	2	3	4	...	i
$y =$	0	1	4	9	?	...	$y(i)$

and students are asked to fill in the result for 4 and to determine a general formula for determining *any* element in the sequence. Next comes an algebra course where students experience the joys of such exciting problems as two trains leaving from Chicago and Philadelphia and colliding in Pittsburgh.

Functional programmers know that these students are encountering their first core concept in their mathematical education: functions and "variable expressions." They also know that functions and expressions don't need to be restricted to numbers and operations on numbers. It is perfectly acceptable to speak of the arithmetic and algebra of booleans, chars, strings, etc.

DrScheme programmers also formulate functions and expressions that compute with images as first-class values, e.g.,

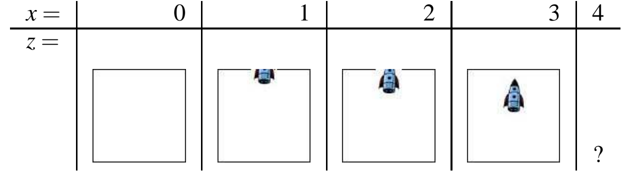
```
(empty-scene 100 100)
```

returns a blank square of 100 by 100 pixels, and

```
(place-image [rocket] 10 20 (empty-scene 100 100))
```

combines the image of the rocket with the blank square by placing the former 10 pixels to the right of the left margin and 20 pixels down from the top margin of the latter.

Now imagine teaching in this context. Teachers can ask students what the next image in the following series is:



and what the general formula is for the image. As before, students would struggle and eventually come up with an answer.

At this point, teachers could explain that displaying 25 to 30 of these scenes per second would create the effect of a "movie" that simulates a rocket landing. Students know movies, and students find movies more interesting than trains colliding in Pittsburgh. So the teacher could show them how to play this movie in DrScheme:

```
(define (rocket-scene i)
  (place-image [rocket] 50 i (empty-scene 100 100)))
```

The function definition captures the general answer to the teacher's question, though in parentheical syntax. Although this isn't close to the historically grown infix notation of mathematics (or to that of fashionable languages), our experience shows that students don't seem to mind after some initial reluctance. If the student now

[1] Due to the demand, Schanzer [personal communication, Dec. 2008] is working on a second-level course for next summer.

[2] Over the past few years, Alice [alice.org] and Scratch [scratch.mit.edu] have been touted as frameworks for teaching middle school students how to program. Both efforts are incomparable with Bootstrap for two reasons. First, both Alice and Scratch are mostly imperative and thus fail to have direct benefits for the students' mathematics education. Second, as others also note (Powers et al. 2007), these GUI-oriented systems come without a natural transition to full-fledged programming whereas our curriculum spells out a natural path from middle school mathematics to the second semester in college.

applies the library function `run-simulation` to `rocket-scene`, the library generates a series of scenes. More precisely, it applies its argument (here: `rocket-scene`) to 0, 1, 2, 3, ... and displays the resulting series of images in a separate window, like the one embedded in this paragraph, at the rate of 28 numbers per second.

The rest of the paper shows how to generalize this idea so that the language of 9th grade school mathematics can be used to design interactive, and even distributed, games.

4. Designing a World

The HtDP curriculum heavily emphasizes functional programming. DrScheme (Findler et al. 2002), HtDP's accompanying IDE, supports a series of five teaching languages, each expanding the expressive power of its predecessor. The first four of these teaching languages are purely functional, and they are usually the only ones used in courses for novice programmers.

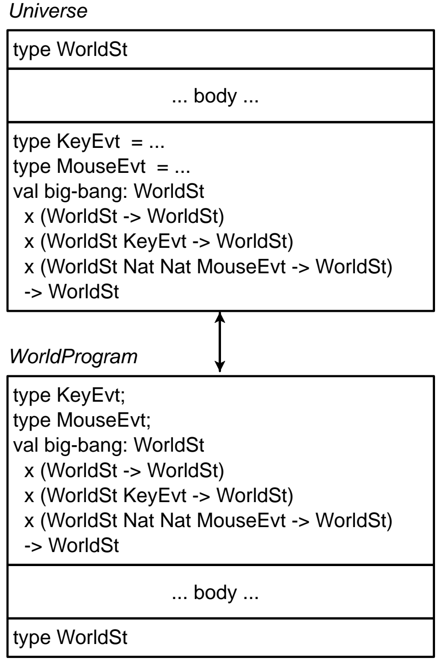


Figure 1. A unit perspective of world programs and UNIVERSE

Our I/O framework comes as a library, dubbed UNIVERSE. The library implements and exports two expression forms for launching *world* and *universe* programs. This section explains worlds; the next one is about universes.

4.1 The World is a Virtual Machine

To a student, the UNIVERSE library represents the computer's operating system and hardware. As such the library is the keeper of a representation of the state of the world. When the hardware or operating system notices certain events, the library hands over the state of the world to a function in the student's program and expects another state back. We call this state a *world*, and the phrase *world program* denotes the collection of functions that interact with the library. Combining the library and a world program creates an executable program.

The library is parameterized over the kinds of states—called `WorldSt`—that the world program wishes to deal with as well as the event handlers that process these states. A world program matches these two parameters with a data definition for the collection of states and with a collection of functions. Figure 1 expresses this dependency between the library and a student's program via a unit diagram (Flatt and Felleisen 1998). The universe library is parameterized over the type `WorldSt`; it exports two types and a function that consumes `WorldSt`-processing functions. Conversely, a world program is a unit that imports all of this, exporting in return a `WorldSt` type. Linking the two creates the executable.

In reality, though, programs specify types as comments, and UNIVERSE does not export a function for specifying event handlers but a syntactic extension, dubbed **`big-bang`**:

```
(big-bang WorldState-expr
          (on-tick tock-expr rate-expr†)†
          (on-key react-expr)†
          (on-mouse click-expr)†
          (stop-when done-expr)†
          (on-draw render-expr width-expr†
                               height-expr†)†)
```

A **`big-bang`** expression has one required sub-expression—the initial state of the world—and five optional clauses (indicated via † superscripts). These clauses are introduced via one of five keywords (`on-tick`, `on-key`, `on-mouse`, `stop-when`, and `on-draw`), mimicking keyword-based parameter passing. Each clause specifies at least one sub-expression; two have additional optional sub-expressions (see † superscripts).

When PLT Scheme encounters a **`big-bang`** expression, it first evaluates all sub-expressions and checks some basic properties. The result of `WorldState-expr` becomes the initial state of the world. The remaining values give access to a subset of the underlying platform's events:

1. If an `on-tick` clause exists, **`big-bang`** starts a clock that ticks at a rate of 28 times per second or as often as the result of `rate-expr`—a natural number—specifies.

   The expression `tock-expr` must evaluate to a function of one argument:

   ```
   ;; WorldSt → WorldSt
   ```

   Specifically, the function consumes a state of the world and produces one. The `universe` library invokes it on the current state every time the clock ticks; its result becomes the next state.

2. An `on-key` clause specifies how a world program reacts to a keyboard event. Its sub-expression must evaluate to a function of two arguments:

   ```
   ;; WorldSt KeyEvt → WorldSt
   ```

   The first is again the current state of the world; the second is a data representation of the keyboard event.

   In UNIVERSE, a keyboard event is represented as either a one-character string (e.g., `"a"`) or a number of special strings (e.g., `"left"`, `"release"`). The former denote regular keys on the keyboard; the latter are used to represent arrow keys, other special keys, and the event of releasing a key.

   The library invokes this function for every keyboard event and uses the result of the invocation as the new state of the world.

3. Similarly, an `on-mouse` clause determines how a world program reacts to a mouse event with a function of four arguments:

   ```
   ;; WorldSt Nat Nat MouseEvt → WorldSt
   ```

   As always, the first argument is the current state of the world. The next two arguments capture the *x* and *y* coordinates of the

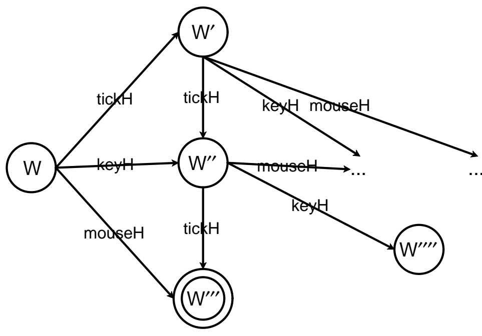


Figure 2. A state transition diagram for world programs

event, measured in the number of pixels from the left and top of the screen. Finally, the `MouseEvt` argument determines what kind of mouse action has taken place. It is one of the following six strings: `"button-up"`, `"button-down"`, `"drag"`, `"move"`, `"enter"`, `"leave"`.

Like the underlying operating system, the UNIVERSE library does not notify a world program of every mouse event, but it samples the mouse events at a reasonably high rate.

The result of applying the mouse-event handler function becomes the next world.

4. The `stop-when` clause determines when the world ends. Its sub-expression must evaluate to a predicate:

```
;; WorldSt → Boolean
```

After handling an event with one of the above event-handling functions, UNIVERSE uses this predicate to find out whether the resulting state of the world is a *final* state. If the result state satisfies the predicate, no further events are processed.

5. Last but not least, a **`big-bang`** expression may come with an `on-draw` clause, which has either one or three sub-expressions. The first sub-expression must evaluate to a function of one argument:

```
;; WorldSt → Image
```

If the **`big-bang`** expression specifies such a function, the UNIVERSE library opens a separate window whose size is determined by the size of the first image that the function produces. Alternatively, a program may specify the size of the canvas explicitly via the two additional sub-expressions, which must evaluate to natural numbers.

The function specified in `on-draw` is used every time an event-handling function produces a state. The resulting image is rendered in the separate window.

Once the world ends, **`big-bang`** returns the final state.[3]

As figure 2 suggests, the core of an executable world program denotes a state machine. Each element `W`, `W`′, `W`″, ... of `WorldSt` is a state of this machine. For each state and for each kind of event, the event handlers (plus event-specific inputs) specify the successor state; that is, each state—except for final ones—is the source of three (family of) arrows (with distinct targets). The final states are those for which the predicate specified in the `stop-when` clause produces `true`.

[3] It is instructive to contrast this to the type of `reactimate` in Fran (Elliot and Hudak 1997).

What the figure does not show is the orthogonally specified rendering of each state as a scene or image. Although these images are values in PLT Scheme, they are usually not a component of world states. One way to imagine this rendering process is to add a different kind of arrow to each state and connecting this arrow to the scene that the `on-draw` function produces for this state.

Given this explanation, we can explain the workings of the `run-simulation` function. Its world is the world of natural numbers, i.e., the state of the world represents the number of times the clock has ticked so far:

```
;; WorldSt = Nat
;; interp. the number of clock ticks
```

As for `run-simulation`, it consumes a function from natural numbers to `Scenes`. Its purpose is to start the world, to count the number of clock ticks, and to invoke the given function on each clock tick to render a series of `Scenes`:

```
;; (Nat → Scene) → Nat
(define (run-simulation render)
  (big-bang 0 (on-tick add1) (on-draw render)))
```

The result of `run-simulation` is a natural number: specifically, the number of clock ticks that have passed (once the simulation halts).

4.2 Designing a World Program

Designing a world program is surprisingly easy. The first step is to design a data representation for the information that varies and that is to be tracked through the duration of the program execution. We recommend expressing the data representation as a data (type) definition (or several) and equipping it with comments that interpret this data in terms of the visible canvas (world). Naturally, this data definition fills in for the `WorldSt` type from the preceding section.

The second step is to tease out constants that describe properties of the world. This includes both quasi-physical constants, e.g., the width and height of the screen, as well as image constants, e.g., the background or a fixed shape that moves across the scenery.

The third step is to design the event-handling functions. Here "design" refers to the design recipe from HtDP. Given that we already have data definitions (from the first step and the library), we also have contracts for all the top-level functions. Hence the next step is to think through examples and to turn them into tests. The creation of templates usually (but not always) uses the `WorldSt` type for orientation. After coding, it is important to run the tests.

Also following HtDP, iterative development is the most appropriate approach for world programs. Specifically, we recommend that students provide a minimally useful data definition for `WorldSt` and then design one state-processing event handler and the rendering function. This enables them to test the core of the program and interact with it. From here, they can pursue two different directions: enriching the data and adding event handlers.

4.3 Controlling a UFO

Let us illustrate how to design world programs with an example from the second or third week in a college freshman course. The goal of the exercise is to move a UFO ("flying saucer") across the canvas in a continuous manner. Later we add functions that allow "players" to control the UFO's movements via the arrow keys on the keyboard and via mouse clicks.

A moving object on a flat canvas has (at least) four properties, meaning we need to use a structure[4] to represent the essential data:

[4] In teaching languages, a structure definition like this one introduces three kinds of functions: a constructor (`make-ufo`), a predicate (`ufo?`), and one selector per field to extract the values (`ufo-x`, `ufo-y`, `ufo-dx`, `ufo-dy`). PLT Scheme also adds imperative mutators on demand.

```
;; WorldSt KeyEvt → WorldSt
;; control the ufo's direction via the arrow keys

(check-expect
  (control (make-ufo 5 8 -1 -1) "down")
  (make-ufo 5 8 -1 +1))
;; ... more test cases ...

(define (control w ke)
  (cond
    [(key=? ke "up") (set-ufo-dy w -1)]
    [(key=? ke "down") (set-ufo-dy w +1)]
    [(key=? ke "left") (set-ufo-dx w -1)]
    [(key=? ke "right") (set-ufo-dx w +1)]
    [else w]))

;; WorldSt Int → WorldSt
(define (set-ufo-dy u dy)
  (make-ufo (ufo-x u) (ufo-y u)
            (ufo-dx u) dy))
```

```
;; WorldSt Nat Nat MouseEvt → WorldSt
;; move the ufo to a new position on the canvas
(check-expect (hyper (make-ufo 10 20 -1 +1)
                     40 30 "button-up")
              (make-ufo 10 20 -1 +1))
;; ... more test cases ...

(define (hyper w x y a)
  (cond
    [(mouse=? "button-down" a)
     (make-ufo x y (ufo-dx w) (ufo-dy w))]
    [else w]))
```

```
;; WorldSt → Boolean
;; has the ufo landed?
(check-expect (landed? (make-ufo 5 (- SIZE 5) -1 +1))
              false)
;; ... more test cases ...

(define (landed? w) (>= (ufo-y w) SIZE))
```

Figure 3. Using keyboard and mouse events to control a ufo

```
(define-struct ufo (x y dx dy))
;; WorldSt = (make-ufo Nat Nat Int Int)
;; interp. the location (pixels)
;;         and velocity (pixels/tick)
```

Because nothing else in this "game" changes over time, we identify the state of the world with the state of the UFO.

Next we fix the size of the canvas, the background (an empty scene), and the shape of the UFO:

```
(define SIZE 400)
(define MT (empty-scene SIZE SIZE))
(define UFO
  (overlay (circle 10 "solid" "green")
           (rectangle 40 2 "solid" "green")))
(define UFO.version2 )
```

This time we use basic image creation and manipulation primitives to create the right kind of shape; using the definition of `UFO.version2` instead of `UFO` would of course work equally well.

With the above data definition, we have determined the complete type signature of the event-handling functions for clock tick events. Of course we should add a purpose statement:

```
;; WorldSt → WorldSt
;; move the ufo for one tick of the clock
```

The next step in our design recipe calls for examples that describe the behavior of the function. We formulate these examples immediately in the unit testing framework that comes with DrScheme's teaching languages:[5]

```
(check-expect (move (make-ufo 10 20 -1 +1))
              (make-ufo 9 21 -1 +1))
```

The example illustrates that the function's purpose is to add the velocity to the current position and to use it as the new position:

```
(define (move w)
  (make-ufo (+ (ufo-x w) (ufo-dx w))
            (+ (ufo-y w) (ufo-dy w))
            (ufo-dx w)
            (ufo-dy w)))
```

[5] DrScheme collects all **check-expect** expressions and evaluates them after all definitions and expressions are evaluated. It then outputs the results and tabulates failed test cases with hyper-links to the source text of the test.

Before we can interact with the program, we must design one more function, namely, a function for rendering the current state of the world as a scene:

```
;; WorldSt → Scene
;; place the ufo into MT at its current position

(check-expect (render (make-ufo 10 20 -1 +1))
              (place-image UFO 10 20 MT))

(define (render w)
  (place-image UFO (ufo-x w) (ufo-y w) MT))
```

Designing such a function proceeds according to the same recipe as designing the `move` function. Also notice that we can test the outcome of this function as if it were a function on the reals. Because images are first-class values, it makes sense to construct the expected output and to compare it to the actual result of the function. PLT Scheme's standard `equal?` function works for images, too. While we recommend that students develop such "expected results" expression (interactively in the REPL) to gain some understanding of how the function should proceed, it is indeed possible to insert an actual image instead of such an expression:

```
(check-expect
 (render (make-ufo 10 20 -1 +1)) )
```

Equipped with `move` and `render`, it is possible to define a main function and to watch these first two definitions in action:

```
;; WorldSt → WorldSt
;; run a complete world program,
;; starting in state w0
(define (main w0)
  (big-bang w0 (on-tick move) (on-draw render)))
```

In short, we have finished the first stage of our iterative design cycle, creating a first useful part of the overall program.

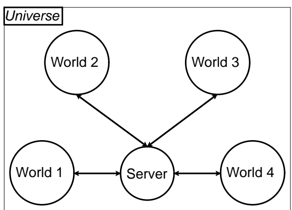
From here, it is easy to design the rest of the function. See the left-hand side of figure 3 for the definition of a function that controls the movements of the UFO via arrow keys. The function `key=?` compares two keyboard events. The right-hand side of the same figure displays functions for making the UFO jump to the position of a mouse click; `mouse=?` of course compares mouse events. The last function checks whether the UFO has landed.

5. Universe: A World is Not Enough

Designing interactive graphical programs via purely functional programming is only half the game. The other half is about designing distributed programs, especially distributed games. The principles remain the same, but the differences deserve a close look.

5.1 Universes

A universe consists of a distributed collection of world programs that communicate with each other via a programmable server:



We make no assumption about where the programs run, in particular, UNIVERSE cannot find servers automatically.

The communication links rely on TCP/IP connections, meaning messages sent from a world to a server (or vice versa) are guaranteed to arrive in the order in which they are dispatched. Of course, when two distinct world programs send messages to the server, there is no guarantee that the messages arrive in the order they were sent; similarly, if the server broadcasts messages to (some of) the participating worlds, the messages may again arrive at distinct worlds in an unrelated order.

In order to design a universe based on the UNIVERSE teachpack, students design a communication protocol, which they implement via a "server" program. Some protocols simply pass messages from one world program to another and back, with the server playing the role of a conduit. Other protocols assume that the server is an arbiter, enforcing the rules of a game or directing traffic among the participants, as in a chat room. Finally, the server could be configured in such a way that the world programs simulate peers in a peer-to-peer neighborhood.

5.2 A World in the Universe

For a world program to participate in a universe, it registers with the server using a `(register ip-expr)` clause in its **big-bang** expression. The sub-expression designates an IP address (as a string).

A registered world program sends messages via its event handlers. To this end, the UNIVERSE library defines `package` structures and exports its constructor and predicate:

```
(define-struct package (world msg))
;; Package = (make-package World S-exp)
```

Moreover, the library actually deals with event handlers that return one of two kinds of results, meaning the signature of, say, key event handlers is really

```
;; WorldSt KeyEvt → (∪ Package WorldSt)
```

instead of the one specified in the preceding section. If an event handler produces a package, the library uses the value in the first field as the next state of the world, and the value in the second field is sent off to the server. Otherwise, the library assumes the result is just the state of the world.

To receive messages, a world program installs an event-handling function via an `on-receive` clause in **big-bang**. It subexpression must evaluate to a function with the following signature:

```
;; WorldSt S-exp → (∪ Package WorldSt)
```

When a message in the form of an S-expression arrives, this event handler is applied to the current state of the world and the message. Like all other event handlers, this handler may return a `Package`.

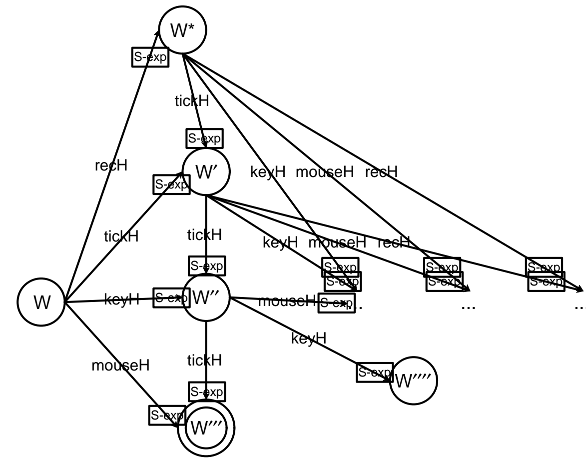


Figure 4. State transition view of world *communicating* programs

Figure 4 is a revision of figure 2 for communicating worlds. Again, all elements of `WorldSt` are states, but now all states come with four kinds of transition arrows. The fourth one is the event handler that deals with message receipts. In addition, each arrow now comes with an optional *output label* in the form of an S-expression. Just as UNIVERSE displays the rendering of a state as an image for a world program, it also implements the sending these messages from state transitions to the universe's server.

5.3 The Universe and its Server

The UNIVERSE library supports the design of *servers* in a manner that is analogous to the design of world programs. A programmer describes a server via a pair of specifications: a data definition of universe states, dubbed `UniSt`, and a **universe** description, which is analogous to a **big-bang** description.

For a server, three kinds of events matter most: the entry of an additional world into the universe, a world's disappearance, and the arrival of a message from a participating world. Accordingly server programs must deal with representations of participating worlds, and UNIVERSE supports this:

```
(define-struct iworld (name in out))
;; IWld = (make-iworld String Port Port)
;; interp. internal representation
;;         of a participating world
```

The `iworld` structure keeps track of a world program's name, its input TCP port, and its output port, though a server program may only access the name field of `iworld` structures. Other than that, server programs must compare worlds and do so with `iworld=?`.

Here is the core grammar of a **universe** description:

```
(universe UniSt-expr
          (on-new new-expr)
          (on-msg msg-expr)
          (on-disconnect disc-expr disc-expr)†
          ...)
```

The first, required sub-expression determines the initial state of the server. Furthermore, every **universe** description comes with an `on-new` clause and an `on-msg` clause. Optionally, it may also contain an `on-disconnect` clause.

Every server's event handler consumes the current state of the universe—as perceived and maintained by the server's event handlers—and the representatin of a participating world; it may also consume a message received from such a world. An event

handler produces a bundle, i.e., a UNIVERSE-specified structure that contains three distinct pieces of information: the new server state (UniSt); a list of messages to designated worlds; and the list of worlds to be discarded:

```
(define-struct bundle (state mails to-discard))
(define-struct mail (to msg))
;; Bundle = (make-bundle UniSt Mail* IWld*)
;; Mail* = [Listof (make-mail IWld S-exp)]
;; IWld* = [Listof IWld]
```

Event handlers may only construct bundles and mails; they may not destructure them.

The event handlers function as follows:

1. An on-new handler has the signature

   ```
   ;; UniSt IWld → Bundle
   ```

   i.e., it consumes the server state and a representation of the world that wishes to join. The resulting bundle may contain this new world as one that should be discarded, which effectively represents a rejection of the request. Optionally, the handler may send out messages about the event.
2. An optional on-disconnect event handler has the same signature as an on-new handler, but it deals with the disappearance of a world from the universe:

   ```
   ;; UniSt IWld → Bundle
   ```

   This kind of event is usually due to a severed connection or because the corresponding world program shut down.
3. The signature for on-msg handlers also includes the message that arrived in the form of an S-expression:

   ```
   ;; UniSt IWld S-exp → Bundle
   ```

   When the on-msg event handler is invoked, it is applied to the state of the server, the world that sent in a message, and the message itself. The result bundle determines how this event is shared with other worlds in the universe.

Optional handlers may drive the server via clock ticks, render the current state of the server in a console, or deal with other events.

A complete universe program—as specified in a **universe** expression—is best thought of as a state-transition machine, just like the one for world programs depicted in figure 4. Each element of UniSt is a state of the machine; each event handler (and its auxiliary parameters) represents one possible transition from one UniSt element to another. In contrast to world programs, the state transitions in a universe program come with two labels: one for sending mail to a list of participating worlds, and another one for deleting worlds from the list of participants.

5.4 Designing a Universe

Designing a universe requires two different perspectives: a global one concerning coordination and local ones for the server and the world programs. Once the global view has been developed, the local design of the servers and world programs proceeds just like stand-alone world programs.

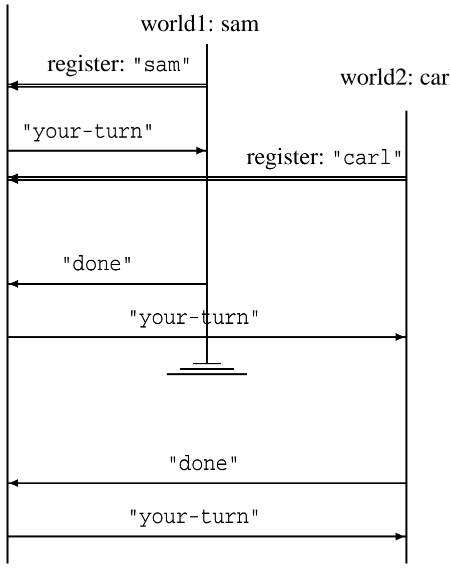
The global perspective demands the design of a coordination and communication protocol. This protocol design has the goal of creating and maintaining an invariant for the universe. In order to achieve this goal, we teach students to consider the start-up phase, the steady-state phase, and the shut-down phase of a universe. For all cases, it is important to understand (1) the order in which events occur and (2) which S-expressions encode which messages.

Our experimentation with the UNIVERSE library suggests that interaction diagrams—like those used for object-oriented designs based on UML—are a good medium for discussing ideas. Instead of spelling out this recommendation in detail, however, we illustrate it with a simple example.

5.5 Serving a Turn

As mentioned, the coordination among the worlds of a universe depends on the server and the message protocol it employs. We and our students have implemented a number of servers. Here we illustrate the power of the UNIVERSE library with the design of a server and some UFO controller clients where each client gets a turn to control a (local) UFO. We start with the protocol design, followed by the design of the server, and then the adaptation of the UFO program from section 4 to support distribution.

Protocol Design The prose suggests the following, informal and schematic interaction diagram:



The three vertical lines are "world life-lines," while the horizontal lines are registration or message sending steps.

This particular diagram shows the key properties of our proposed universe. The server is on the left; the participating worlds are to its right. After creation, a world registers with the server, which we assume sends along a name for the world. Our diagram shows that as soon as a first world has registered, the server gives this world a turn without waiting for any other world to show up. If another world shows up—possibly during some turn—the server becomes aware of it but continues to wait for a "done" signal from the world whose turn it is. Once the active world ends its turn, the server gives a turn to the next world on the list. Finally, the diagram also shows what happens when a world disappears, say due to the closure of a connection. The server notes the disappearance and gives a turn to (one of) the remaining worlds.

Server Design From here, the design of the server proceeds just like the design of a world program, though we must observe the constraints imposed by the protocol. We start with the required data definition:

```
;; UniSt = IWld*
;; interp. list of worlds in the order they take
;;         turns, starting with the active one
;;         the active world (if any) is first
```

```
;; UniSt IWld → Bundle
;; nw is joining the universe
(check-expect
 (add-world (list iworld2) iworld1)
 (make-bundle (list iworld2 iworld1) '() '()))
;; ... more test cases ...
(define (add-world ust nw)
  (if (empty? ust)
      (make-bundle (list nw) (m2 nw) '())
      (make-bundle (append ust (list nw)) '() '())))
```

```
;; UniSt IWld "done" → Bundle
;; mw sent message m; assume mw = (first ust), m = "done"
(check-expect
 (switch (list iworld1 iworld2) iworld1 "done")
 (make-bundle (list iworld2 iworld1) (m2 iworld2) '()))
;; ... more test cases ...
(define (switch ust mw m)
  (local ((define l (append (rest ust) (list mw)))
          (define nxt (first l)))
         (make-bundle l (m2 nxt) '())))
```

```
;; UniSt IWld → Bundle
;; dw disconnected from the universe
(check-expect
 (del-world (list iworld1 iworld3) iworld3)
 (make-bundle (list iworld1)  '() '()))
;; ... more test cases ...
(define (del-world ust dw)
  (if (not (iworld=? (first ust) dw))
      (make-bundle (remq dw ust) '() '())
      (local ((define l (rest ust)))
        (if (empty? l)
            (make-bundle '() '() '())
            (local ((define nxt (first l))
                    (define mll (m2 nxt)))
              (make-bundle l mll '()))))))
```

```
;; IWld → Mail*
;; create single-item list of mail to w
;; no test cases
(define (m2 w)
  (list (make-mail w "your-turn")))
```

Figure 5. A primitive functional server program

Note again interpretation that comes with the data definition. It has several implications for the design of the event handlers.

Since this server deals with three kinds of events—registration of a world, message receipt, and disconnection of a world from the universe—we need three event handlers. The UNIVERSE specifications and the agreement to send certain messages dictate the contract statements:

```
;; add-world : UniSt IWld → Bundle
;; switch : UniSt IWld "done" → Bundle
;; del-world : UniSt IWld → Bundle
```

The names of the three functions are suggestive of their purpose.

Just as in the case of the UFO controller, we can design these functions in a systematic manner. In support of unit tests for event handlers in a server, UNIVERSE exports three sample worlds `iworld1`, `iworld2`, and `iworld3`; of course, it does not export the capability of creating representations of participating worlds. Otherwise, the design of these three server functions proceeds in a straightforward fashion.

The three definitions and fragments of their test suites are displayed in figure 5:[6]

1. the top-left box contains the code for adding a world;
2. the box in the bottom-left defines the function for dealing with a message from the active world, which is the only kind of messages that the server expects;
3. the top-right box concerns the event of a world disconnecting from the universe; and
4. the final box in the bottom right contains the definition of a auxiliary function for creating a list of mail to a single world.

As far as the server is concerned, the only task left to do is to formulate the **`universe`** expression and to evaluate it at DrScheme's reply to start the server:

[6] The definitions use the **`local`** construct from the HtDP teaching languages. Roughly speaking, `(local defs body)` introduces the mutually recursive definitions `defs` for the evaluation of `body`. Unlike Scheme's internal definitions, **`local`** definitions have the exact same semantics as global definitions but come with a restricted lexical scope.

```
(universe '()
          (on-new add-world)
          (on-msg switch)
          (on-disconnect del-world))
```

Adding this expression to the bottom launches a process that waits for TCP/IP events and deals with them by invoking one of the three event handlers.

Client Design To illustrate how the client side works, let us consider a small change to our UFO controller from the preceding section. Suppose we give each "player" a turn to land a UFO and that when the UFO touches the ground, it is the next world's turn. One obvious implication is that there is now a distinct new kind of state of the world:

```
;; WorldSt is one of:
;; --- "rest"
;; --- (make-ufo Nat Nat Int Int)
```

When it isn't this world's turn, the world is in a `"rest"` state.

Next we replace the event handler for ticks with a function that sends out messages when the UFO lands:

```
;; WorldSt → (∪ WorldSt Package)
(define (move.global w)
  (cond
    [(string? w) w]
    [else (local ((define v (move w)))
            (if (not (landed? v))
                v
                (make-package "rest" "done")))]))
```

The function distinguishes the two cases from the data definition. For a string, it returns the world as is. Otherwise, it moves the world using the old `move` function and then checks whether the UFO has landed; if so, the new event handler produces a package.

In addition, we need a handler for `"your-turn"` messages:

```
;; WorldSt "your-turn" → WorldSt
;; assume: messages arrive only
;;        if the state is "rest"
(define (receive w msg)
  (make-ufo 20 10 -1 +1))
```

```
;; WorldSt (WorldSt → WorldSt) (WorldSt KeyEvt → WorldSt) (WorldSt Nat Nat MouseEvt → WorldSt) [Listof Event]
;; →
;; WorldSt
;; process a list of events given the initial world and event handlers
(define (big-bangF w0 tickH keyH mouseH loe0)
  (local (... dispatch: see below, on the right ...
          ;; accumulator design: w is the result of dealing with all events between loe0 and loe (inclusive)
          (define (big-bangF w loe)
            (cond
              [(empty? loe) w]
              [else (big-bangF (dispatch w (first loe)) (rest loe))])))
    (big-bangF w0 loe0)))
```

```
(define-struct tick ())
(define-struct key (kind))
(define-struct mouse (x y kind))
;; An Event is one of:
;; --- (make-tick)
;; --- (make-key KeyEvt)
;; --- (make-mouse Nat Nat MouseEvt)
```

```
;; WorldSt Event → WorldSt
;; deal with a single event, given the state of the world
(define (dispatch w e)
  (cond
    [(tick? e) (tickH w)]
    [(key? e) (keyH w (key-kind e))]
    [(mouse? e) (mouseH w (mouse-x e) (mouse-y e) (mouse-kind e))]))
```

Figure 6. The semantics of functional event handling

Unlike `move.global`, `receive` does not distinguish two kinds of worlds. Whether the world is in a resting state or not, the function returns some UFO.

The revised `main` function registers the world with the server and specifies a name for the world that is used for registration:

```
;; String → WorldSt
(define (main-for-client n)
  (big-bang "rest"
            (on-tick move)
            (on-draw renderR)
            (on-rec receive)
            (name n)
            (register LOCALHOST)))
```

Here we assume the server is running on the same computer as the client and that `renderR` renders the new kind of worlds.

Note: The design assumes that all participating worlds and the server implement the protocol correctly. The assumptions above suggest how functions may protect themselves against errors in the implementations or attacks. The reader may wish to explore the small changes needed to check those assumptions.

6. Design and Curriculum

Designing reactive programs in a purely functional manner comes with several advantages. For one, it is straightforward to explain **big-bang** as if it were a function. As figure 6 shows, this function traverses a list of events,[7] accumulating the changes to the initial world. Also, it uniquely fits in with our design curriculum, which covers functional design followed by courses on logical reasoning and object-oriented design.

6.1 Design Recipe

HtDP introduces its teaching programming languages as a generalization of school mathematics. Instead of functions over just numbers, these languages can express functions and expressions that deal with atomic data (numbers, symbols, chars, strings, images, boolean data) and compound data (structures, vectors, and lists). In the third and fourth part of the book (and its teaching languages) **lambda** and local definitions are added.

Programming is developed as the systematic design of computational solutions to "word" problems. The design of individual functions follows a general six-step procedure paired with a systematic development of data definitions. The design of programs is presented as an iterative refinement process, comparable to the scientific process of developing models of the world. Specifically the program is the model, and the world is the set of our (or our client's) objectives. As we refine the program, our model satisfies more and more of the objectives.

Obviously, this design recipe also applies to the design of I/O functions for world and universe programs. The key is that UNIVERSE translates external information into internal data and invokes the event handlers on the latter. Furthermore, the event handlers produce only internal data, which UNIVERSE then displays as external information. The translations are hidden from the students' transformations. Hence, the process of formulating contracts, functional examples, etc. remains the same. Because images are just another form of atomic data, the design recipe even applies to the rendering functions that produce complex graphical scenes.

The separation of the actual act of performing I/O from the processing or production of I/O data is critical for effective testing. It empowers a programmer to unit-test every single function, covering the complete chain from where input data appears to the point of where output data is delivered. As a matter of fact, this covers the testing of image-producing functions for which we recommend two different testing strategies. The first is to develop an expression in the read-eval-print loop of DrScheme that creates an image for simple inputs. This kind of experimentation suggests both an "expected value" expression as well as the body for the desired function. The second strategy is to create the expected image separately:

```
(check-expect (create-ufo) ━●━)
(check-expect (render-world (make-ufo ...))
  (place-image ━●━
               ... ...
               (empty-scene SIZE SIZE))
```

As the second **check-expect** specification shows, it is of course possible to mix and match those two strategies.

Once tests are developed, DrScheme's built-in test coverage tool pin-points those expressions that haven't been evaluated during a

[7] Our implementation replaces the list with an imperative stream of events, plus a thread for receiving messages from the server. The stream dispatcher and the thread are coordinated via the CML-inspired synchronization primitives of PLT Scheme.

```
(define world%
  (class fun-world% (super-new)
    (init-field ufo)
    (field [MT (empty-scene 500 500)])
    ;; → world%
    ;; deal with a tick event in this world
    (define/augment (tick)
      (new world% [ufo (send ufo move/tick)]))
    ;; → scene<%>
    ;; render this world as a scene
    (define/augment (render)
      (send ufo render MT))))
```

```
(define world%
  (class imp-world% (super-new)
    (init-field ufo)
    (field [MT (empty-scene 500 500)])
    ;; → void
    ;; deal with a tick event in this world
    (define/augment (tick)
      (send ufo move/tick))
    ;; → scene<%>
    ;; render this world as a scene
    (define/augment (render)
      (send ufo render MT))))
```

```
(define ufo%
  (class object% (super-new)
    (init-field x y dx dy)
    (field [UFO (overlay (rectangle ...) (circle ...))])
    ;; → ufo%
    ;; move this ufo for one tick
    (define/public (move/tick)
      (new ufo% [x (+ x dx)][y (+ y dy)][dx dx][dy dy]))
    ;; → scene<%>
    ;; add this ufo to the given scene s
    (define/public (render s) (place-image UFO x y s))))
```

```
(define ufo%
  (class object% (super-new)
    (init-field x y dx dy)
    (field [UFO (overlay (rectangle ...) (circle ...))])
    ;; → void
    ;; effect: change this ufo's coordinates, for a move
    (define/public (move/tick)
      (begin (set! x (+ x dx)) (set! y (+ y dy))))
    ;; → scene<%>
    ;; add this ufo to the given scene s
    (define/public (render s) (place-image UFO x y s))))
```

Figure 7. Applicative and imperative world classes

test run. We want novice programmers to attempt to cover all expressions, except for those that connect the event handlers to the underlying operating system (**big-bang**, **universe**). While complete coverage is a good first goal, the design of reactive programs tends to demonstrate that unit testing does not suffice. Even when an individual reactive function passes all unit tests, the composition of all the reactive functions to deal with a large stream of events often concocts scenarios that the unit tests don't cover. Put differently, reactive programming demands some amount of integration testing, too. Given our "list of events" semantics, programmers can usually mimic these scenarios with the composition of event handlers.

Last but not least, because the event handlers are just functions, we can also subject them to the functional random testing (Claessen and Hughes 2000) tools now built into DrScheme or its theorem proving environment (Eastlund 2009). Indeed, programmers who learn to formulate conjectures and validate conjectures via random testing are ideally prepared to study the automated verification of interactive/reactive programs.

6.2 Reasoning about Worlds and Universes

During their second semester at Northeastern University, computer science majors study the logic of computation. The course combines a standard theoretical introduction into logic with practical hands-on exercises based on the ACL2 system (Boyer and Moore 1996); see our experience report on the test run of this course (Eastlund et al. 2007). Roughly speaking, the ACL2 system consists of an applicative Common Lisp and an automatic theorem prover based on first-order classical logic.

Two years ago we extended the ACL2 system with the UNIVERSE library, enabling students to write reactive games, formulate conjectures about the safety of their game programs, and prove them correct via the ACL2 theorem prover (Eastlund and Felleisen 2009). Here is a typical theorem from such experiments:

```
(defthm preserve-safety
  (implies (safe-state game-state)
           (safe-state (tick game-state))))
```

When the theorem prover fails, students are encouraged to subject their conjectures to our ACL2 random tester (Eastlund 2009).

The mechanized proofs are based on the semantics of the **big-bang** function in figure 6 and a more general version for universes of world programs. Specifically, a macro unfolds claims about a specific instance of **big-bang** expressions into an application of a function like `big-bangF` to all possible lists of events.

6.3 On to Classes

At the same time as freshmen learn to formulate claims about their functional animation programs and to prove them correct, they are enrolled in a parallel course on design in the context of class-based object-oriented languages. We prepare the transition at the end of the first semester with some simple conventions and arrangements. Specifically, instead of arranging functions by feature (e.g., all rendering functions in one place, all key-event related functions somewhere else), we organize functions around data definitions.

For example, we start with all event handlers for `WorldSt`:

```
;; WorldSt is one of ...

;; WorldSt → WorldSt
(define (world-tickh w) ...)

;; WorldSt → Scene
(define (world-render w) ...)

;; WorldSt KeyEvt → WorldSt
(define (world-keyh w ke) ...)
```

and follow it up with an arrangement around `UFO`:

```
;; UFO is one of ...

;; UFO → UFO
(define (ufo-move u) ...)

;; UFO Scene → Scene
(define (ufo-add-to-scene u s) ...)

;; UFO Symbol → UFO
(define (ufo-chg u dir) ...)
```

We always make the current state the first parameter of a function, analogous to the implicit *this* parameter in methods.

An experienced programmer can immediately see that programming functional I/O methods is notationally even more convenient in a class-based context than in a functional language. In contrast to functions, methods are defined in a context where all the pieces of a world are accessible as fields.

Consider the left-hand side of figure 7. It displays a version of the UFO program in PLT Scheme's class system (Flatt et al. 2006).[8] The functions from section 4 have been turned into methods of a class `world%` and `ufo%`. Each event-handling method returns a new instance of the class. Instead of selectors, the methods use field names to access the current values of the world state. Furthermore, the `world%` class is derived from an abstract class that provides default functionality for all event handlers and the imperative functionality for connecting event-handling methods to the machine's devices. It naturally motivates inheritance and overriding.

Finally, while an applicative world design with classes is notationally superior to a structure-based design, it still suffers from the notational overhead of creating new objects for every transformation. The `move/tick` ("move per tick") method in `ufo%`, for instance, copies both the `dx` and the `dy` field into the new instance. Compare this method with `move/tick` in the imperative variant of `ufo%` on the right-hand side of figure 7. In general, the transition from a state-transforming functional program to an imperative object-oriented program is straightforward, easy to explain, and thus clarifies to students how the design principles of their first, functional experience carries over to the languages they expect to encounter in college.

7. Related Work

From a technical perspective, the Clean Event I/O system (Achten and Plasmeijer 1995) comes closest to our approach.[9] The Clean programming language supports so-called abstract I/O devices to which programs attach event handlers. In contrast to our event handlers, a Clean event handler has the following signature

```
;; WorldSt × *DeviceSt → WorldSt × *DeviceSt
```

where `DeviceSt` type represents the state of an abstract I/O device. The $*$ notation on a type adds a linearity constraint on the type; the type system enforces this linearity constraints for the matching function parameter. For event handlers, the linearity constraint means that reading and writing to the I/O devices is enabled and translated into efficient imperative actions. Naturally, linearity constraint also has implications for the design and organization of event handlers, making them look like imperative functions.

Our I/O framework supports only devices (windows, keyboards, mouse clicks, clocks) whose state can be supplied all at once when an event handler is invoked. Conversely, if a state needs to change, the event handlers don't write to the device. Instead, the library uses an orthogonal rendering function to translate the state into an image that it displays, or it allows event handlers to return an additional value that it writes to a TCP port. In short, because our framework completely decouples event processing from writing to a device, there is no need in our framework to use linearity types and to thread the state of a device through an event handler.

An additional difference between Clean and UNIVERSE concerns the nature of the devices. In Clean I/O devices are abstract types; in UNIVERSE the rendering functions translate states into concrete types (images). This concreteness enables UNIVERSE programmers to test all functions of an interactive graphical program, including those that produce output. Contrast this situation with the use of an abstract device type in Clean and of the I/O monad in Haskell. The testing of I/O functions in such a framework is similar to the testing of imperative procedures, requiring elaborate set-up and tear-down steps. We consider this activity out of reach for middle school students and distracting for courses that focus on design.

Functional reactive programming (FRP) (Elliot and Hudak 1997) overcomes this problem by enabling programmers to write in a functional style over imperative values (event streams, behaviors). The programmer effectively describes a dataflow graph via expression dependencies; the run-time system updates values using this graph. While programming with event streams and behaviors is truly elegant, our pedagogic experience has been that the necessity of operators like `switch` puts it out of the reach of novices. Technically, FRP also has the disadvantage of requiring devices to be adapted to behave as reactive elements, which is a research problem that has been solved only partially (Ignatoff et al. 2006).

Erlang (Armstrong et al. 1996) factors its I/O framework in a different but related manner. A distributed program in Erlang also consists of world-transforming event handlers, though such a program also need a process-local loop to keep track of the state. Our UNIVERSE library naturally separates these two concerns by factoring out the common loop from the server and the participants.

From a pedagogical perspective, van Dam and his colleagues (1987, 1995) pioneered the event-oriented approach for teaching novices in the 1980s, but via imperative object-oriented programming. Bruce et al. (2001, 2004) resumed this direction in the early 2000s. We consider the functional alternative presented here even more useful than an imperative, object-oriented approach. On one hand, a functional approach is close to the mathematics that students encounter, meaning our approach promises a straightforward skill transfer. While we have only anecdotal evidence so far, we are convinced that a formal evaluation would confirm this conjecture. On the other hand, we consider object-oriented programming for novices an overkill because beginners don't have programs of enough complexity to benefit from the structuring that object-orientation provides and demands.

Chakravarty and Keller (2004) share our analysis concerning the teaching of functional programming languages in the first course as well as the problems of Haskell I/O. Their reaction is to turn this weakness of Haskell into an advantage. Specifically, the course switches perspective, emphasizing the imperative character of I/O *actions* and the need for *ordering actions*. While we acknowledge the pedagogical need for a transition to imperative programming, we consider this strategy a kludge and prefer the systematic approach via objects explained in section 6.3. After all, postponing I/O suggests that functional programming can't cope with the full spectrum of programming tasks and fails to exploit it for the motivational aspects of assignments.

An alternative and appealing solution is due to Achten (2008), who packaged up one special-purpose case study (playing soccer) along the lines of our framework. Sadly focusing on soccer limits the appeal of the framework to certain cultures and countries.

Finally, Hudak and Peterson each briefly taught Haskell-based functional programming to small groups of selective middle school and high school students. Both arranged lectures around Haskore and Pan but did not use any texts [Hudak and Peterson, independent personal communication, Feb. 2009.]

8. Conclusion

Our work demonstrates that with a suitable I/O framework, purely functional programming is an engaging medium for students of all ages. The Bootstrap effort routinely guides middle school students

[8] In our courses and workshops, we use Java.

[9] Acten (with Weirich, 2000) turned the Clean Event I/O system into the Clean Object I/O system and later ported it to Haskell (Achten and Jones 2001). Daan Leijen provided a binding to the wx media kit, now known as the wxHaskell toolkit [Achten, personal communication, Feb. 2009].

without apparent mathematical talent to write interactive games in a language that is basically equivalent to high school algebra. For freshman students, we exploit the same framework to simultaneously strengthen their mathematical skills and to introduce them to the basics of program design. In one second-semester course, students even use an automatic theorem prover to establish interesting properties about such interactive games. At the same time, event-driven programming can also be used to prepare freshmen for a course on object-oriented programming.

Our work relies on two key insights and one technicality. First, it is important to leave the translation of external information into internal data (structures) to the framework and vice versa. As far as students are concerned, these are tasks that the computer and/or the operating systems takes on for the program. Second, the framework must separate event handling (as state transitions) from rendering (from states to images, sounds, or message transmission). This separation of concerns empowers novice programmers to design one function per task, without worrying about ordering any computational actions. One DrScheme-specific technicality facilitates the second step: turning images into first-class values. Although inserting images into programs and dealing with them directly at an interactive read-eval-print can be especially helpful, we don't expect this technicality to be critical for an adaptation of our approach to other functional languages. In short, we conjecture that every functional language can easily supplement its I/O system with a library such as ours and could thus become an appealing medium for a range of educational applications.

Acknowledgments We gratefully acknowledge the help of many people: Carl Eastlund for feedback on the design and for discussions concerning its logical content; Kathi Fisler for using experimental releases of the library in her courses; Emmanuel Schanzer for creating and coordinating the Bootstrap outreach program; and Danny Yoo for extending the library with hierarchical GUI features.

References

Peter Achten. Teaching functional programming with soccer-fun. In *Proc. 2008 International Workshop on Functional and Declarative Programming in Education*, pages 61–72, 2008.

Peter Achten and Simon L. Peyton Jones. Porting the Clean object I/O library to Haskell. In *IFL '00: Selected Papers from the 12th International Workshop on Implementation of Functional Languages*, pages 194–213, London, UK, 2001. Springer-Verlag.

Peter Achten and Marinus J. Plasmeijer. The ins and outs of Clean I/O. *J. Funct. Program.*, 5(1):81–110, 1995.

Peter Achten and Martin Wierich. A tutorial to the Clean Object I/O library (version 1.2). Technical report, University of Nijmegen, February 2000.

Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang (2nd Edition)*. Prentice-Hall, 1996.

Bird and Wadler. *Introduction to Functional Programming (2nd Edition)*. Prentice Hall PTR, 1998.

Robert S. Boyer and J Strother Moore. Mechanized reasoning about programs and computing machines. In R. Veroff, editor, *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*, pages 146–176. The MIT Press, Cambridge, Massachusetts, 1996. URL `citeseer.ist.psu.edu/boyer96mechanized.html`.

Kim B. Bruce, Andrea Danyluk, and Thomas P. Murtagh. Event-driven programming is simple enough for cs1. *SIGCSE Bull.*, 33(3):1–4, 2001.

Kim B. Bruce, Andrea Danyluk, and Thomas P. Murtagh. Event-driven programming facilitates learning standard programming concepts. In *Object-oriented programming systems, languages, and applications: Educators Symposium*, pages 96–100, 2004.

Manuel Chakravarty and Gabriele Keller. The risks and benefits of teaching purely functional programming in first year. *J. Funct. Program.*, 14(1): 113–123, 2004.

Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ACM SIGPLAN International Conference on Functional Programming*, pages 268–279, 2000.

Carl Eastlund. DoubleCheck your theorems. In *Proc. 8th Intern. Works. ACL2 and its Applications*, pages 41–46. Lulu Press, 2009.

Carl Eastlund and Matthias Felleisen. Automatic verification for interactive graphical programs. In *Proc. 8th Intern. Works. ACL2 and its Applications*, pages 33–41. Lulu Press, 2009.

Carl Eastlund, Dale Vaillancourt, and Matthias Felleisen. ACL2 for freshmen: First experiences. In *Proc. 7th Intern. ACL2 Symposium*, pages 200–211. ACM Press, 2007.

Conal Elliot and Paul Hudak. Functional reactive animation. In *ACM SIGPLAN International Conference on Functional Programming*, pages 196–203, 1997.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to Design Programs*. MIT Press, 2001. URL `http://www.htdp.org/`.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The TeachScheme! project: Computing and programming for every student. *Computer Science Education*, 14:55–77, 2004a.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. The structure and interpretation of the computer science curriculum. *J. Funct. Program.*, 14(4):365–378, 2004b.

Robert Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *J. Funct. Program.*, 12(2):159–182, March 2002.

Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 236–248, June 1998.

Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Scheme with classes, mixins, and traits. In *Asian Symposium on Programming Languages and Systems (APLAS) 2006*, pages 270–289, November 2006.

Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge Univ. Press, 2000.

Graham Hutton. *Programming in Haskell*. Cambridge Univ. Press, 2007.

Daniel Ignatoff, Gregory H. Cooper, and Shriram Krishnamurthi. Crossing state lines: Adapting object-oriented frameworks to functional reactive languages. In *International Symposium on Functional and Logic Programming*, pages 259–276, 2006.

Bryan O'Sullivan, Donald Stewart, and John Goerzen. *Real World Haskell*. O'Reilly Media, Inc., 2008.

Kris Powers, Stacey Ecott, and Leanne Hirshfield. Through the looking glass: teaching CS0 with Alice. *SIGCSE Bulletin*, 39(1):213–217, 2007.

Simon Thompson. *Haskell: the Craft of Functional Programming*. Addison Wesley Longman Publishing Co., Inc., 1997.