All You Need Are Functions

A Brief Introduction to Functional Programming in Haskell

SPGS 14 November 2016

Henrik Nilsson

Functional Programming Laboratory, School of Computer Science

University of Nottingham, UK

Outline

- Why programming language research?
- What is functional programming and how is it different?
- A Taste of Haskell: A Pure, Lazy, Functional Language
- Some real-world examples (games!)

The Functional Programming Lab (1)

What do we do?

The Functional Programming Lab (1)

What do we do?

Programming language research, with a focus on functional languages, into:

The Functional Programming Lab (1)

What do we do?

Programming language research, with a focus on functional languages, into:

- Foundations: Underpinning mathematical principles

The Functional Programming Lab (1)

What do we do?

Programming language research, with a focus on functional languages, into:

- Foundations: Underpinning mathematical principles
- Language Design

The Functional Programming Lab (1)

What do we do?

Programming language research, with a focus on functional languages, into:

- Foundations: Underpinning mathematical principles
- Language Design
- Applications

The Functional Programming Lab (1)

What do we do?

Programming language research, with a focus on functional languages, into:

- Foundations: Underpinning mathematical principles
- Language Design
- Applications

These inform one another.

The Functional Programming Lab (2)

Why?

The Functional Programming Lab (2)

Why?

- It's scientifically interesting! For example, some of the foundational research touches on the very foundations of mathematics itself.

The Functional Programming Lab (2)

Why?

- It's scientifically interesting! For example, some of the foundational research touches on the very foundations of mathematics itself.
- Want to make it easier to write better programs.

The Functional Programming Lab (2)

Why?

- It's scientifically interesting! For example, some of the foundational research touches on the very foundations of mathematics itself.
- Want to make it easier to write better programs.

Better?

The Functional Programming Lab (2)

Why?

- It's scientifically interesting! For example, some of the foundational research touches on the very foundations of mathematics itself.
- Want to make it easier to write better programs.

Better? Many aspects, including:

The Functional Programming Lab (2)

Why?

- It's scientifically interesting! For example, some of the foundational research touches on the very foundations of mathematics itself.
- Want to make it easier to write better programs.

Better? Many aspects, including:

- Fewer (preferably no!) software errors or "***bugs***".

The Functional Programming Lab (2)

Why?

- It's scientifically interesting! For example, some of the foundational research touches on the very foundations of mathematics itself.
- Want to make it easier to write better programs.

Better? Many aspects, including:

- Fewer (preferably no!) software errors or "***bugs***".
- More reusable.

The Functional Programming Lab (2)

Why?

- It’s scientifically interesting! For example, some of the foundational research touches on the very foundations of mathematics itself.
- Want to make it easier to write better programs.

Better? Many aspects, including:

- Fewer (preferably no!) software errors or “***bugs***”.
- More reusable.
- More maintainable.

The Cost of Software Errors

The Cost of Software Errors

- Cambridge University study (2012):
  - $312 ***billion***
  - Half of development effort spent on finding and fixing errors

The Cost of Software Errors

- Cambridge University study (2012):
  - $312 ***billion***
  - Half of development effort spent on finding and fixing errors
- Google estimates of cost per bug:
  - Unit test: $5
  - Full build: $50
  - Integration test: $500
  - System test: $5000

The Cost of Software Errors

- Cambridge University study (2012):
  - $312 ***billion***
  - Half of development effort spent on finding and fixing errors
- Google estimates of cost per bug:
  - Unit test: $5
  - Full build: $50
  - Integration test: $500
  - System test: $5000

The cost of bugs that make it into “the wild”?

(In)famous Bugs (1)

(In)famous Bugs (1)

- 1985–1987: Therac-25: Radiation therapy machine. At least six fatal overdoses. Bug occurred very rarely, slowing its discovery.

(In)famous Bugs (1)

- 1985–1987: Therac-25: Radiation therapy machine. At least six fatal overdoses. Bug occurred very rarely, slowing its discovery.
- 1996: First test flight of Ariane 5 failed with rocket self-destructing, including $500-million satellite payload. Cause: numerical overflow.

(In)famous Bugs (1)

- 1985–1987: Therac-25: Radiation therapy machine. At least six fatal overdoses. Bug occurred very rarely, slowing its discovery.
- 1996: First test flight of Ariane 5 failed with rocket self-destructing, including $500-million satellite payload. Cause: numerical overflow.
- 1998: NASA's $665-million Mars Climate Orbiter fails to enter orbit. Burns in Mars's atmosphere instead.
  Reason?

(In)famous Bugs (1)

- 1985–1987: Therac-25: Radiation therapy machine. At least six fatal overdoses. Bug occurred very rarely, slowing its discovery.
- 1996: First test flight of Ariane 5 failed with rocket self-destructing, including $500-million satellite payload. Cause: numerical overflow.
- 1998: NASA's $665-million Mars Climate Orbiter fails to enter orbit. Burns in Mars's atmosphere instead.
  Reason? Someone forgot to convert from imperial to metric units.

(In)famous Bugs (2)

- 2015: 3200 US prisoners released on average 49 days early due to software glitch. System had been in operation since 2002.

(In)famous Bugs (2)

- 2015: 3200 US prisoners released on average 49 days early due to software glitch. System had been in operation since 2002.
- 2015: Starbucks point-of-sales systems down, making it impossible to accept payment. Many happy customers get drinks for free. Cost to Starbucks: A few million dollars.

(In)famous Bugs (2)

- 2015: 3200 US prisoners released on average 49 days early due to software glitch. System had been in operation since 2002.
- 2015: Starbucks point-of-sales systems down, making it impossible to accept payment. Many happy customers get drinks for free. Cost to Starbucks: A few million dollars.

Many and diverse reasons for failures: no one solution. But better programming language technology could have prevented some; e.g. the Mars orbiter crash.

Declarative Programming (1)

Wikipedia:

> Declarative programming is a programming paradigm [style] that expresses the logic of a computation without describing its control flow.

Declarative Programming (1)

Wikipedia:

> Declarative programming is a programming paradigm [style] that expresses the logic of a computation without describing its control flow.

To put this differently: more ***what*** (logic), less ***how*** (control).

Declarative Programming (2)

How can that help?

Declarative Programming (2)

How can that help?

- Clearer, more concise programs (as fewer details to worry about).

Declarative Programming (2)

How can that help?

- Clearer, more concise programs (as fewer details to worry about).
- Easier to ***prove*** programs correct.

Declarative Programming (2)

How can that help?

- Clearer, more concise programs (as fewer details to worry about).
- Easier to ***prove*** programs correct.

Functional Programming is a type of declarative programming where programs are built ***exclusively*** from functions and function application.

Declarative Programming (2)

How can that help?

- Clearer, more concise programs (as fewer details to worry about).
- Easier to ***prove*** programs correct.

Functional Programming is a type of declarative programming where programs are built ***exclusively*** from functions and function application.

In particular, functions in the basic mathematical sense: ***equational reasoning*** is applicable.

List of Squares: Python (1)

```
def squares(m,n):
    ss = []
    for i in range(m, n + 1):
        ss.append(i * i)
    return ss
```

List of Squares: Python (1)

```
def squares(m,n):
    ss = []
    for i in range(m, n + 1):
        ss.append(i * i)
    return ss

>>> squares(1,5)
```

List of Squares: Python (1)

```
def squares(m,n):
    ss = []
    for i in range(m, n + 1):
        ss.append(i * i)
    return ss

>>> squares(1,5)
 ???
```

List of Squares: Python (1)

```
def squares(m,n):
    ss = []
    for i in range(m, n + 1):
        ss.append(i * i)
    return ss

>>> squares(1,5)
[1, 4, 9, 16, 25]
```

List of Squares: Python (2)

```
def squares(m,n):
    ss = []
    for i in range(m, n + 1):
        ss.append(i * i)
    return ss
```

Note:

- Step-by-step description of the algorithm: explicit ***control flow***; "***how***".
- The result list is constructed one element at a time.

List of Squares: Haskell

```
squares m n
    | m > n     = []
    | otherwise = m*m : squares (m+1) n

> squares 1 5
[1, 4, 9, 16, 25]
```

Note:

- Direct statement of ***what*** the list of squares is.
- Recursion.
- The result list is expressed as a whole.

Other differences: Function Types

Python:

```
>>> type(squares)
<type 'function'>
```

`squares` is a function, but we're not told what the types of its arguments and result are.

Other differences: Function Types

Python:

```
>>> type(squares)
<type 'function'>
```

`squares` is a function, but we're not told what the types of its arguments and result are.

Haskell:

```
> :type squares
squares :: (Num a,Ord a) => a -> a -> [a]
```

For any numeric type `a`, `squares` is a function from two numbers of type `a` returning a list of numbers of the ***same*** type `a`.

Other differences: Polymorphism

Python:

```
>>> squares(1.0, 5.0)
```

Other differences: Polymorphism

Python:

```
>>> squares(1.0, 5.0)
```

???

Other differences: Polymorphism

Python:

```
>>> squares(1.0, 5.0)
 TypeError: range() integer end
argument expected, got float.
```

Other differences: Polymorphism

Python:

```
>>> squares(1.0, 5.0)
 TypeError: range() integer end
argument expected, got float.
```

Haskell:

```
> squares 1.0 5.0
[1.0, 4.0, 9.0, 16.0, 25.0]
```

Other differences: Polymorphism

Python:

```
>>> squares(1.0, 5.0)
 TypeError: range() integer end
argument expected, got float.
```

Haskell:

```
> squares 1.0 5.0
[1.0, 4.0, 9.0, 16.0, 25.0]
```

The Haskell version of `squares` is ***polymorphic***, or “of many shapes”: in this case, works for ***any*** numeric type as all we assumed was multiplication and addition.

Dynamic vs. Static Typing (1)

Python:

```
def foo():
    return squares([2,3,5,7])
```

Dynamic vs. Static Typing (1)

Python:

```
def foo():
    return squares([2,3,5,7])
```

???

Dynamic vs. Static Typing (1)

Python:

```
def foo():
    return squares([2,3,5,7])
 >>>
```

The definition of `foo` is accepted!

Dynamic vs. Static Typing (1)

Python:

```
def foo():
    return squares([2,3,5,7])
 >>>
```

The definition of `foo` is accepted!

```
>>> foo()
```

Dynamic vs. Static Typing (1)

Python:

```
def foo():
    return squares([2,3,5,7])
 >>>
```

The definition of `foo` is accepted!

```
>>> foo()
```

???

Dynamic vs. Static Typing (1)

Python:

```
def foo():
    return squares([2,3,5,7])
 >>>
```

The definition of `foo` is accepted!

```
>>> foo()
 TypeError: squares() takes exactly
2 arguments (1 given)
```

The error only caught when we attempt to run `foo`.

Dynamic vs. Static Typing (2)

Haskell:

```
> foo () = squares [(2::Int),3,5,7]
No instance for (Num [Int])
```

The error caught immediately: essentially we are told that a list of integers is not a number.

Dynamic vs. Static Typing (2)

Haskell:

```
> foo () = squares [(2::Int),3,5,7]
No instance for (Num [Int])
```

The error caught immediately: essentially we are told that a list of integers is not a number.

Static typing certainly not unique to functional languages. But some of the most sophisticated type systems have been developed for functional languages.

Equational Reasoning (1)

Python:

```
a = 10
def fie(n):
    return a * n
>>> fie(2)
```

Equational Reasoning (1)

Python:

```
a = 10
def fie(n):
    return a * n
>>> fie(2)
```

???

Equational Reasoning (1)

Python:

```
a = 10
def fie(n):
    return a * n
>>> fie(2)
 20
```

Equational Reasoning (1)

Python:

```
a = 10
def fie(n):
    return a * n
>>> fie(2)
 20
```

Thus, `fie(2) = 20`. Right?

Equational Reasoning (1)

Python:

```
a = 10
def fie(n):
    return a * n
>>> fie(2)
 20
```

Thus, `fie(2) = 20`. Right?

But what about:

```
a = 20
fie(2)
```

Equational Reasoning (1)

Python:

```
a = 10
def fie(n):
    return a * n
>>> fie(2)
 20
```

Thus, `fie(2) = 20`. Right?

But what about:

```
a = 20
fie(2)
```

???

Equational Reasoning (1)

Python:

```
a = 10
def fie(n):
    return a * n
>>> fie(2)
 20
```

Thus, `fie(2) = 20`. Right?

But what about:

```
a = 20
fie(2)
 40
```

Equational Reasoning (2)

Thus, in Python, `fie` is not a function in the usual mathematical sense. It is not ***pure***.

Equational Reasoning (2)

Thus, in Python, `fie` is not a function in the usual mathematical sense. It is not ***pure***.

In contrast, ***Haskell***:

```
> let a = 10
> let fie n = a * n
> let a = 20
> fie 2
20
```

Equational Reasoning (2)

Thus, in Python, `fie` is not a function in the usual mathematical sense. It is not ***pure***.

In contrast, ***Haskell***:

```
> let a = 10
> let fie n = a * n
> let a = 20
> fie 2
20
```

`fie 2` = `20` always! We can replace `fie 2` by `20` or vice versa anywhere without changing the meaning of a program. This is what is meant by ***equational reasoning***.

Equational Reasoning (3)

Why is it (arguably) a practical advantage to program with pure functions?

Equational Reasoning (3)

Why is it (arguably) a practical advantage to program with pure functions?

A pure function has a simple, well-defined ***interface***: its meaning is independent of context and calling it does not cause any ***side effects***. As a consequence, much easier to:

- Understand large programs
- Reuse code
- Reason about code

Try Haskell (1)

Point your browser to `http://tryhaskell.org`.

- A string in Haskell is the same as a list of characters. I.e.

  ```
  ['a', 'b', 'c'] = "abc"
  ```

  Try it: type in `['a', 'b', 'c']` to verify.
- Try functions `head`, `tail`, `reverse`, `sort` on your name. E.g. `head "Henrik"`. What do they do?
- Write an expression that extracts:
  - The second letter of your name
  - The last letter of your name

Try Haskell (2)

- What is `[1..10]`?
- Write an expression for the list of all integers from 50 to 100.
- Do `head`, `tail`, `reverse` work on lists of numbers?
- What is the type of `head`, `tail`, `reverse`?
  Hint: just type in e.g. `head` and hit return.
  What do the types mean?
- What does the function `sum` do to a list of numbers?
- Write an expression to sum all integers from 1 to 1000.

Try Haskell (3)

- `(*2)` is a function that multiplies a number by 2; `(^2)` is a function that squares a number. Try!
- `map` is a ***higher order*** function: it takes a function as an argument and applies it to every element in a list. Explain the result of:
  - map (*2) [1..10]
  - map (^2) [1..10]
- Sum the squares from 1 to 1000.
- What does `words` do to your full name?
- Extract the initials from your full name.

Infinite Data Structures (1)

Haskell is a ***lazy*** functional language: nothing is evaluated unless ***needed*** (and then at most once).

This makes it possible to program with (conceptually) ***infinite*** data structures, such as lists.

Infinite Data Structures (1)

Haskell is a ***lazy*** functional language: nothing is evaluated unless ***needed*** (and then at most once).

This makes it possible to program with (conceptually) ***infinite*** data structures, such as lists.

More generally, laziness promotes declarative programming. It allows us to focus more on “what”, less on “how”, as there is less need to worry about exactly when things get computed: they get computed automatically as and when needed.

Infinite Data Structures (2)

Given:

```
ones = 1 : ones

from n = n : from (n + 1)

nats = from 0
```

we have

```
> take 10 ones
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
> take 10 nats
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The Sieve of Eratosthene

The following defines `primes` to be the list of ***all*** prime numbers!

```
sieve (p : xs) =
  p : sieve [ x | x <- xs, x `mod` p /= 0 ]

primes = sieve (from 2)
```

The 10 first and the 10000th prime number:

```
> take 10 primes
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
> primes !! 9999
104729
```

So, What About *Real* Programs . . .

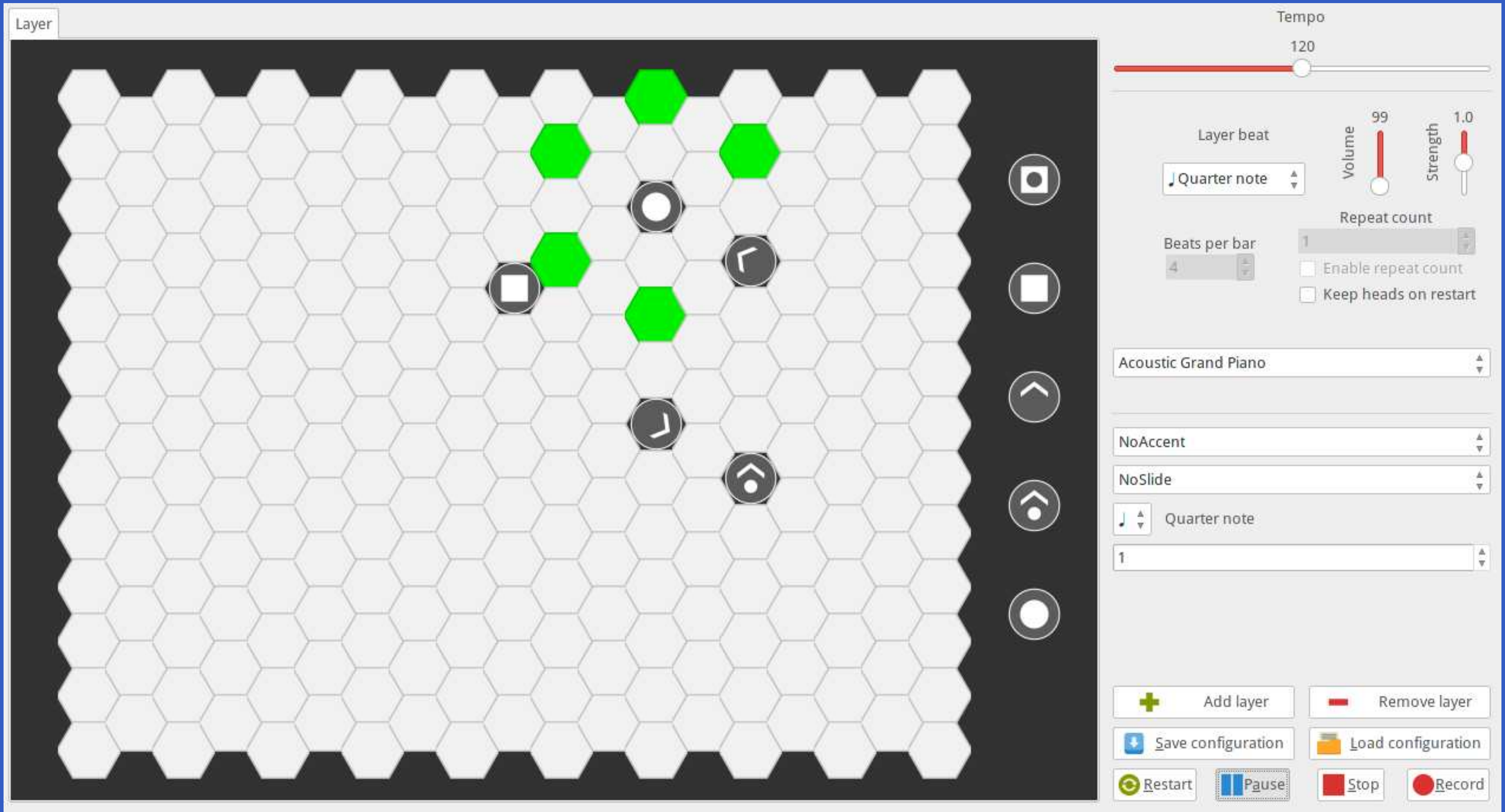
So, What About *Real* Programs . . .

. . . like ***games***?

So, What About *Real* Programs ...

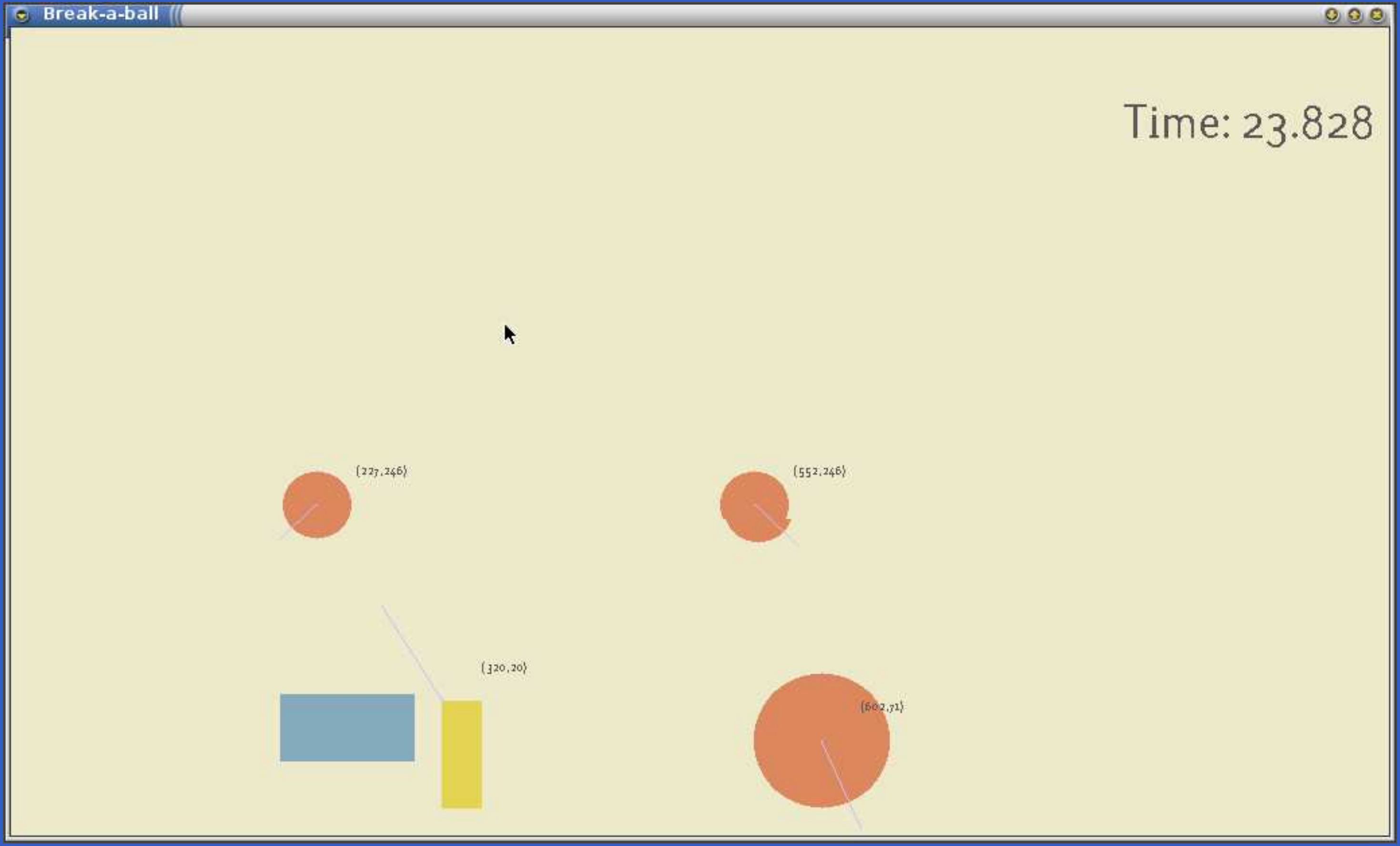
... like ***games***?

Or Musical Applications?



Take-home Game!

Download for free to your Android device!



Play Store: Pang-a-lambda (Keera Studios)

But How???

How can we even think about games, musical applications, etc. as pure functions? What about interaction?

But How???

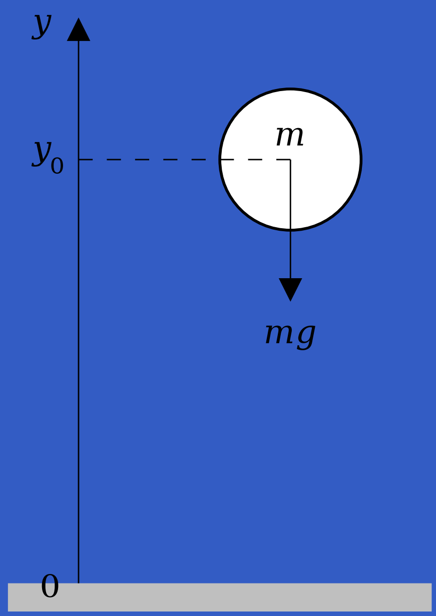
How can we even think about games, musical applications, etc. as pure functions? What about interaction?

One possibility: pure functions on ***signals*** or ***time-varying values***:

- Player input
- Video output
- Input from a musical keyboard
- Notes to be played on a synthesizer
- Audio output

A Bouncing Ball

Lots of bouncing balls in Pang-a-lambda!



$$\begin{aligned} y &= y_0 + \int v \,\mathrm{d}t \\ v &= v_0 + \int -9.81 \end{aligned}$$

On impact:

$$v = -v(t-)$$

(fully elastic collision)

Mathematical equations that describe a falling ball: a simple ***physical model***.

Modelling a Free-falling Ball

```
type Pos = Double
type Vel = Double

fallingBall :: Pos -> Vel -> SF () (Pos, Vel)
fallingBall y0 v0 = proc () -> do
    v  <-  (v0 +) ^<< integral  -<  -9.81
    y  <-  (y0 +) ^<< integral  -<  v
    returnA -< (y, v)
```

Some different and extra symbols, but just superficial syntactic details: the ***structure*** remains the same. We have turned the mathematical model into a declarative program!

More information

- `http://www.haskell.org`
- John Hughes, recent retrospective: Why Functional Programming Matters `https://www.youtube.com/watch?v=FGQAP0Gx1W8`