Bricklayer: An Authentic Introduction to the Functional Programming Language SML

Victor Winter
Outline

- Motivation
- Educational Challenges
- Bricklayer
In Practice

- Multi-core and many-core technologies are on the rise.
  - The New Moore’s Law

- Two Models of Concurrency
  - Shared state
  - Message passing
Shared State Concurrency

- **Mutable State**
  - Mutex or synchronized method needed to lock a specific region in memory
- **Threats:**
  - Deadlock and starvation
  - What happens when a program in possession of a lock crashes?
- **Suited to “Von Neumann isomorphic” models of computation**
  - Java, C++
Message Passing Concurrency

- Immutable State (stateless)
- Well-suited to computational models underlying functional programming languages

“Your Erlang program should just run N times faster on an N-core processor.”

— Joe Armstrong
Java 8

- Lambda expressions and method references
  - Can be passed as parameters to other methods
- Essential to multi-core programming
  - Applications involving bulk-data on multi-core environments
- Enable Collection APIs to control the parallelization of iterations over the data it stores.
In Education

- Teaching programming and computational thinking needs to occur earlier
  - Syntax-free interfaces
    - Scratch and Alice
  - Authentic environments
    - CodeSpells (Java)
    - Media Computation (Jython)
    - Bricklayer (SML)
In 2011, CMU began teaching functional programming to its freshmen

“Object-oriented programming is eliminated entirely from the introductory curriculum, because it is both anti-modular and anti-parallel by its very nature, and hence unsuitable for a modern CS curriculum.”
Teaching SML

Introductory Concepts
Elements of SML

- Primitive types and operators
- Functions as Values
- Expressions
  - Arithmetic, logical
  - Let-blocks, conditional expressions, expressions sequences
- Definition
  - Variables
  - Non-recursive
  - Recursive
  - Scope
Elements of SML

- Primitive types and operators
- Functions as Values
- Expressions
  - Arithmetic, logical
  - Let-blocks, conditional expressions, expressions sequences
- Definition
  - Variables
  - Non-recursive
  - Recursive
- Scope

primary challenges lie here
Study = (example → problem ) *

- Example:
  1. Problem statement
  2. Problem solution
  3. Computational thinking used to solve the problem

- Examples and problems should be
  - Suitably coupled (conceptually near)
  - Cover a targeted set of learning goals
  - Engaging to students
Incremental Learning

The learning space is actually multidimensional
Effective teaching requires a domain that is...

example rich
and
problem dense
General Educational Challenges

On teaching first-time programmers
The Nature of Computation

- *Fixed-length* computation sequences
  - Converting Celsius to Fahrenheit

- *Variable-length* computation sequences
  - Sorting
Constraints on Programming Assignments

- Concise problem statement
- May not use complex data representations or complex language constructs
- Math problems are often seen as suitable candidates
Bricklayer

An avenue to a domain that is example rich and problem dense
Why LEGO?

- *Engaging* – LEGO as a broad and strong appeal to students and parents
- *Determined* – a LEGO mindset includes construction sequences involving hundreds and even thousands of pieces
- *Concrete* – LEGO constructions have physical manifestations
Overview

- Will be freely available (Summer/Fall 2014)
  http://faculty.ist.unomaha.edu/winter/Bricklayer/index.html

- Modules
  - Basic (New – not included in this presentation)
  - Predicate
  - Brick Functions
  - Basic Navigation
  - Advanced Navigation (under construction)
Predicates

show : cube_side * predicate * brick_type → unit
slideShow : cube_side * predicate * brick_type → unit

xor : cube_side * predicate * brick_type * predicate * brick_type → unit
union : cube_side * predicate * brick_type * predicate * brick_type * brick_type → unit
intersection : cube_side * predicate * predicate * predicate * brick_type → unit
difference : cube_side * predicate * predicate * predicate * brick_type → unit
compliment : cube_side * predicate * predicate * brick_type → unit
```haskell
fun cube n = 
  let
    fun predicate (x,y,z) = true
  in
    Predicate.show(n, predicate, Pieces.AQUAROUND)
  end;

cube 20;
```
Brick Functions

\[
\text{show} : \text{cube\_side} \ast \text{brick\_function} \rightarrow \text{unit}
\]
fun bigCheckerboard boardSize squareSize =
  let
    (* assumes there exists an integer c such that boardSize = c * squareSize *)
    fun equivalenceClass v = (v div squareSize) mod 2

    fun brickFunction(x,y,z) =
      let
        val blue = equivalenceClass x = equivalenceClass z
        in
          if y = 0 then
            if blue then Pieces.BLUE else Pieces.RED
          else Pieces.EMPTY
          end
        in
          BrickFunction.show(boardSize, brickFunction)
        end
  end;

bigCheckerboard 20 5;
Basic Navigation

\[
\begin{align*}
\text{build} & : \text{int} \times \text{int} \times \text{int} \rightarrow \text{unit} \\
\text{show} & : \text{unit} \rightarrow \text{unit} \\
\text{access} & : \text{point} \rightarrow \text{Pieces.brick} \\
\text{update} & : \text{Pieces.brick} \rightarrow \text{point} \rightarrow \text{unit} \\
\text{line} & : \text{point} \rightarrow \text{point} \rightarrow \text{Pieces.brick} \rightarrow \text{unit} \\
\text{traverseWithin} & : \text{point} \rightarrow \text{point} \rightarrow \text{brick_function} \rightarrow \text{unit} \\
\text{traverseXYZ} & : \text{brick_function} \rightarrow \text{unit} \\
\text{traverseXY} & : \text{brick_function} \rightarrow \text{int} \rightarrow \text{unit} \\
\text{traverseXZ} & : \text{brick_function} \rightarrow \text{int} \rightarrow \text{unit} \\
\text{traverseYZ} & : \text{brick_function} \rightarrow \text{int} \rightarrow \text{unit}
\end{align*}
\]

Here is where curried functions first appear.
fun cube n brick = 
    let 
        fun drawLine (x,y,z) = if x < n then ( BasicNavigation.update brick (x,y,z); drawLine(x+1,y,z) ) else () 
        fun drawSquare( y, z ) = if z < n then ( drawLine (0,y,z); drawSquare (y,z+1) ) else () 
        fun drawCube y = if y < n then ( drawSquare (y,0); drawCube (y+1) ) else () 
    in 
        BasicNavigation.build(n,n,n); drawCube 0; BasicNavigation.show() 
    end; 

cube 20 Pieces.GREEN;
Other Examples of Basic Navigation
Advanced Navigation

(under construction)
The End