Programming Skills:
Functional Programming and Haskell

Research Paper Presentations
Research Paper Presentations

Wednesday (December 18)

- Safe Haskell (Haskell’12)
  Sarthak Gupte and Daniyal Iqbal

- The Yampa Arcade (Haskell’03)
  Satyanarayan Iyengar and Nalin Ranjan

- AutoBench: Comparing the Time Performance of Haskell Programs (Haskell’18)
  Justin Lad and Shon Sanchez

- Algebraic Graphs with Class (Functional Pearl) (ICFP’17)
  Zach Morgan and Matthew Richmond

- Functional Array Streams (FHPC’15)
  Nikita Kuchipudi

- Bridging the GUI Gap with Reactive Values and Relations (Haskell’15)
  Elijah Bendinsky and Aj Nagashima

- A meta-EDSL for Distributed Web Applications (Haskell’17)
  Fahad Atif and Henry Farr

- Proving Type Class Laws for Haskell (TFP’19)
  Maheen Riaz Contractor and James Coombs

- Backpack: Retrofitting Haskell with Interfaces (POPL’14)
  Jonathan Patten and Adam Spindler

- There is No Fork: An Abstraction for Efficient, Concurrent, and Concise Data Access (ICFP’14)
  Michael Brice and Daniel Londono Osvath

- Pattern Synonyms (ICFP’16)
  Sergey Goldobin and Josh Spangler
Functional programming combines the flexibility and power of abstract mathematics with the intuitive clarity of abstract mathematics.

https://xkcd.com/1270/
Research Paper Presentations

- Safe Haskell (Haskell’12)  
  Sarthak Gupte and Daniyal Iqbal

- The Yampa Arcade (Haskell’03)  
  Satyanarayan Iyengar and Nalin Ranjan

- AutoBench: Comparing the Time Performance of Haskell Programs (Haskell’18)  
  Justin Lad and Shon Sanchez

- Algebraic Graphs with Class (Functional Pearl) (ICFP’17)  
  Zach Morgan and Matthew Richmond

- Functional Array Streams (FHPC’15)  
  Nikita Kuchipudi

- Bridging the GUI Gap with Reactive Values and Relations (Haskell’15)  
  Elijah Bendinsky and Aj Nagashima
Safe Haskell
Authors: David Terei, David Mazières, Simon Marlow, Simon Peyton Jones

Presented by: Daniyal Iqbal and Sarthak Gupte

Link: https://www.researchgate.net/publication/254462930_Safe_haskell
Agenda

● Overview
● Problems
● Contribution
● Results
● Assignment
Safe Haskell Overview

- Language extension of the Haskell language
- Implemented on GHCI 7.2
- Closes loopholes in Haskell that can bypass typing and module encapsulation
- Allows confining and safely executing untrusted or possibly malicious code
- Unobtrusive, works alongside regular Haskell code
- Enforces properties that Haskell programmers already meet by convention
- Allows for Haskell to become a secure programming language
Problems

- Loopholes to break Haskell properties: Type safety, referential transparency and module encapsulation
- Unsafe language features
- Unsafe language extension
- Guarantee safety of Haskell code execution
Unsafe Language Features

- Can result in unexpected answer b/c unsafely coerce a Float to an Int and add

\[
f :: \text{Int} \rightarrow \text{Float} \rightarrow \text{Int}
\]

\[
f x y = x + \text{unsafeCoerce} \ y
\]

- Non deterministic function call

\[
\text{usec} :: \text{IO Integer}
\]

\[
\text{usec} = \text{getPOSIXTIME} \gg= \text{return} . \text{truncate} . (1000000 * )
\]

\[
f x = x + \text{unsafePerformIO} \ \text{usec}
\]

\[
(\text{let} \ \{ x = f \ 3 \} \ \text{in} \ x + x) \quad (f \ 3 + f \ 3)
\]
Unsafe Language Extensions

- Feature “Generalised Newtype Deriving”
- Can be used to break the module boundary abstraction (encapsulation)

```haskell
module MinList (
    MinList, newMinList, insertMinList
  ) where

data MinList a = MinList a [a] deriving Show

newMinList n = MinList n []

insertMinList s@(MinList m xs) n |
  | n > m = MinList m (n:xs)
  | otherwise = s
```
Example

{-# LANGUAGE GeneralizedNewtypeDeriving #-}  
module Main where

import MinList

class IntIso t where
  intIso :: c t → c Int

instance IntIso Int where
  intIso = id

newtype I = I Int deriving (Eq, IntIso)

-- we reverse the usual comparison order
instance Ord I where
  compare (I a) (I b) = compare b a

nums = [1,4,0,1,-5,2,3]

goodList :: MinList Int
  goodList = foldl insertMinList
          (newMinList $ head nums)
          (tail nums)

badList :: MinList Int
  badList = intIso $ foldl (λx y → insertMinList x $ I y)
          (newMinList $ I $ head nums)
          (tail nums)

main = do
  print goodList
  print badList

MinList 1 [3,2,4]
MinList 1 [-5,0]
Trust

- Method for guaranteeing safety
- Combination of safe and unsafe modules
- Example:
  
  Data.ByteString uses unsafePerformIO to implement a safe interface using mutable byte arrays. Unsafe internals but safe interface. This is OK because it is trusted. Similar to how lazy evaluation is unsafe, but compiler is trusted.

- Method for specifying trust is needed.
Contribution

1. Safe Language Guarantees
2. Identifying Safe Haskell Code
3. Trust
4. Restricted Language Features
Safe Language Guarantees

The Safe Haskell provides certain guarantees, if a code is compiled in a safe language.

- Type Safety - Code block is type safe.
- Referential Transparency - Code block is deterministic.
- Module Encapsulation - If a module is exported, boundaries are strictly enforced. Such that data type can only use functions exported.
- Modular Reasoning - On adding new import module, code independent of it is unaffected by the imported module.
- Semantic Consistency - Meaning of the code block is preserved.
Identifying Safe Haskell Code

- Granularity for Safety is at Module level.
- Classified as Safe, Trustworthy, or Unsafe
- Safe: module is written in safe language
- Trustworthy: not written in safe but author claims that clients can use it safely
- Unsafe: all other modules
- Explicitly specify class: {-# LANGUAGE Safe #-}
Trust

- In Safe Haskell, compiler checks that modules code is safe and uses no unsafe language features, and any imports are trusted.
- Trusted
  1. If module was declared to be Safe and all of imports are safe
  2. User trusts M: Package P is trusted, module is declared to be Trustworthy, all direct safe imports are trusted.
Restricted Language Features

- Unsafe functions are not allowed in safe language. For example - unsafePerformIO

- Completely restricted extensions -
  - Template Haskell - This allows access to the symbols of the modules irrespectively of export list define by the module.
  - Generalised Newtype Deriving as explained using MinList.
  - RULES - This allows changing the semantic of existing code.

- Restricting functionality of extensions to use them with safe language.
  - Foreign Function Interface - Allows import of nonIO type functions.
  - Deriving Data.Typeable - This allows defining one’s own typeable. Language prevents this by allowing automatically derived instances only.
  - Overlapping Instances - Allows redfining of an existing code. Language restricts this such that instance of a module can be redefined by any other instance of that module.
Results

<table>
<thead>
<tr>
<th>Modules</th>
<th>% of base</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safe</td>
<td>23</td>
</tr>
<tr>
<td>Trustworthy</td>
<td>59</td>
</tr>
<tr>
<td>Split</td>
<td>5</td>
</tr>
<tr>
<td>Unsafe</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 1. Results of using Safe Haskell in GHC base package.

<table>
<thead>
<tr>
<th>Modules</th>
<th>% of Hackage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Safe Inferred</td>
<td>3,985</td>
</tr>
<tr>
<td>Unsafe Inferred</td>
<td>10,660</td>
</tr>
</tbody>
</table>

Table 2. Results of inferring Safe Haskell status for all modules on Hackage.
Table 3. Results of compiling all packages on Hackage with a modified Safe pragma (no import restrictions).

<table>
<thead>
<tr>
<th>Modules</th>
<th>Modules</th>
<th>% of failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generalised Newtype Deriving</td>
<td>146</td>
<td>54.48%</td>
</tr>
<tr>
<td>Template Haskell</td>
<td>84</td>
<td>31.34%</td>
</tr>
<tr>
<td>Hand written Typeable instances</td>
<td>33</td>
<td>12.31%</td>
</tr>
<tr>
<td>Non-IO FFI imports</td>
<td>33</td>
<td>12.31%</td>
</tr>
</tbody>
</table>

Table 4. Count of packages that failed to build with a modified Safe pragma (no import restrictions) summed by the language extension that caused the failure.
Use Cases: Secure Programming

{-# LANGUAGE Trustworthy #-}

module PluginAPI (
    RIO(), runRIO, rioReadFile, rioWriteFile
  ) where

-- Notice that symbol UnsafeRIO is not exported
-- from this module!
newtype RIO a = UnsafeRIO { runRIO :: IO a }

instance Monad RIO where
  return = UnsafeRIO . return
  (UnsafeRIO m) >>= k = UnsafeRIO $ m >>= runRIO . k

-- Returns True iff access is allowed to file name
pathOK :: FilePath -> IO Bool
pathOK file = {- Implement some security policy -}

rioReadFile :: FilePath -> RIO String
rioReadFile file = UnsafeRIO $ do
  ok ← pathOK file
  if ok then readFile file else return ""

rioWriteFile :: FilePath -> String -> RIO ()
rioWriteFile file contents = UnsafeRIO $ do
  ok ← pathOK file
  if ok then writeFile file contents else return ()
Assignment

- Create a server that uses Safe Haskell to provide sandboxing for multiple users.
- Simple interface to
  1. Specify the current user
  2. Allow for compiling and executing Haskell code
- Each user can access only their files (files they created), keep track of filenames
- Use RestrictedIO function
- Haskell code can contain malicious code (tries to read/write to files w/out permission), import untrusted libraries..
- Server should only execute if it is trusted (specify modules trusted by sever)
- Code should be written in the safe language and compiled with Safe Haskell, separate pure and IO code and follow rules of Haskell
Questions
The Yampa Arcade

Antony Courtney, Henrik Nilsson, and John Peterson

Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell
Haskell '03, pages 7–18, New York, NY, USA, 2003. ACM.

DOI: 10.1145/871895.871897

Nalin Ranjan and Satyanarayan Iyengar
Motivation

The paper addresses the problem of developing simulated worlds using Functional Reactive Programming (FRP).

FRP integrates the flow of time into purely functional programming languages in a uniform manner.

The authors aim to prove that FRP has the right level of functionality for use in this application domain.
The paper addresses the lack (at the time of publication) of good, practical examples of FRP-style code which solve a complex problem.

It describes an implementation of the arcade classic Space Invaders using Yampa, the authors’ implementation of FRP in Haskell.

The game is made up of reactive objects, which behave in certain ways in response to user input and the passage of time.
Contribution

Yampa promotes an incremental development strategy where individual components are developed and tested in isolation, and then integrated with a larger system.
Signals

A signal is a function from time to a value.

\[ \text{Signal } \alpha \approx \text{Time} \rightarrow \alpha \]

For example, if \textbf{Point} is a type describing a 2-dimensional point, then the position of a mouse pointer over time can be represented as a value of type

\[ \text{Signal Point} \]
Signal Functions

A signal function (SF) is a function that maps one signal to another.

$$\text{SF } \alpha \beta \approx \text{Signal } \alpha \rightarrow \text{Signal } \beta$$

Yampa provides primitive combinators to compose signal functions, forming arbitrary signal function networks.

A Yampa program can be visualized as a network of signal functions, some of which may be active at any time.
data Event a = NoEvent | Event a

switch :: SF a (b, Event c) -> (c->SF a b) -> SF a b

reactimate :: IO (DTime, a)   -- sense
              -> (b -> IO ())   -- actuate
              -> SF a b
              -> IO ()
Space Invaders

Reactive objects:

➔ Gun
➔ Missiles
➔ Invaders
Example - Gun (Definition)

```haskell
data SimpleGunState = SimpleGunState {
    sgsPos :: Position2,
    sgsVel :: Velocity2,
    sgsFired :: Event ()
}

type SimpleGun = SF GameInput SimpleGunState
```
Example - Gun (Control System)

simpleGun :: Position2 -> SimpleGun
simpleGun (Point2 x0 y0) = proc gi -> do
  (Point2 xd _) <- ptrPos -< gi
  rec
    -- Controller
    let ad = 10 * (xd - x) - 5 * v

    -- Physics
    v <- integral -< clampAcc v ad
    x <- (x0+) ^<< integral -< v

  fire <- leftButtonPress -< gi
  returnA -< SimpleGunState {
    sgsPos = (Point2 x y0),
    sgsVel = (vector2 v 0),
    sgsFired = fire
  }
Example - Gun (Testing)

renderGun :: SimpleGunState -> G.Graphic
renderGun = …

gunTest :: IO ()
gunTest = runGame (simpleGun >>> arr renderGun)

runGame is defined using reactimate and IO actions that read input events and render the gun on the window.
Assignment - Flocking System

Flocking:
Group motion behavior where every individual (called a boid) moves independently, but the group as a whole has emergent behavior.

Simulate this behavior using Yampa.

https://medium.com/@michael.bruner3/touchdesigner-flocking-boids-c34dca49afc2
Boid Behavior

At each time step, each boid needs to react to its neighbors and adjust its motion based on the following behaviors:

➔ Cohesion
➔ Separation
➔ Alignment
Cohesion

A boid will try to move towards the centroid of its neighbors.

Craig Reynolds (1987)
Separation

A boid will try to avoid colliding with its neighbors.

Craig Reynolds (1987)
ALIGNMENT

A boid will try to match the average velocity of its neighbors.

Craig Reynolds (1987)
Objective

Simulate the movement of boids in a 2D space (a plane).

The following parameters should be inputs to the system:

➔ Number of boids
➔ Radius of the neighborhood of each boid
➔ Weights for each kind of behavior

The boids can be rendered as primitive shapes. The research paper (*The Yampa Arcade*) can be used as a reference for rendering objects using the Haskell Graphics Library (HGL).
AutoBench: Comparing Time Performance of Haskell Programs

By Martin Handley & Graham Hutton

Presented by: Justin Lad & Shon Sanchez

Motivation & Use Case

- **Correctness**
  - QuickCheck

- **Time Performance**
  - Criterion

- **Solution: AutoBench**
  - Combines QuickCheck and Criterion into single, easy to use library

- **Use Case**
  - Allow average Haskell user to easily measure correctness and time efficiency of functions
Basic Example

slowRev :: [Int] → [Int]
slowRev [] = []
slowRev (x : xs) = slowRev xs ++ [x]

fastRev :: [Int] → [Int]
fastRev xs = go xs []
  where
    go [] ys = ys
    go (x : xs) ys = go xs (x : ys)
Example Graphical Output
## Example Table Output

<table>
<thead>
<tr>
<th>Input Size</th>
<th>0</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>slowRev (μs)</td>
<td>14.00</td>
<td>16.00</td>
<td>19.00</td>
<td>22.00</td>
<td>23.00</td>
<td>...</td>
</tr>
<tr>
<td>fastRev (μs)</td>
<td>14.00</td>
<td>16.00</td>
<td>18.00</td>
<td>19.00</td>
<td>21.00</td>
<td>...</td>
</tr>
<tr>
<td>slowRev</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y = 2.91e−5 + 2.28e−11x + 5.28e−9x²</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fastRev</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y = 1.65e−5 + 2.53e−7x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Optimisation</td>
<td>slowRev ≳ fastRev (0.95)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
QuickCheck Overview

- Generates test inputs of various size and types
- In general, requires defining an *Arbitrary* type to generate test inputs
- AutoBench enables user configurations available in QuickCheck
  - E.g. low & high bounds of input size, and step amount
Criterion Overview

- Haskell library to measure time performance of functions
- Produces statistics including
  - Time performance
  - Line of best fit
  - Comparison of which function is faster
Criterion Types

- Requires defining a *Benchmarkable* type
  - Forcing lazy evaluation (nf)
  - Allowing lazy evaluation (whnf)

\[
\text{nf} :: \text{NFData } b \Rightarrow (a \to b) \to a \to \text{Benchmarkable}
\]
\[
\text{whnf} :: (a \to b) \to a \to \text{Benchmarkable}
\]

- *nf* and *whnf* require a function to be evaluated, \( f :: a \to b \), and an argument \( x :: a \), where \( x :: a \) is generated from quickCheck.
Criterion Types

- By default, time to generate QuickCheck input values is included in time performance.

- To ignore input data time, use the “gen” version of nf or whnf.

\[
\begin{align*}
genNf & : (\text{Arbitrary } a, \text{NFData } a, \text{NFData } b) \Rightarrow \\
& (a \to b) \to \text{Int} \to \text{String} \to \text{Benchmark} \\
\text{genWhnf} & : (\text{Arbitrary } a, \text{NFData } a) \Rightarrow \\
& (a \to b) \to \text{Int} \to \text{String} \to \text{Benchmark}
\end{align*}
\]
Criterion Statistical Analysis

- Computes line of best fit to best describe the function’s time performance
- Uses up to Nth order polynomial functions
- Prone to overfitting
  - Ridge Regression to bound the polynomial coefficients
- Computes monte-carlo cross validation thousands of times to ensure correctness

\[
\text{minimise } \sum_{i=0}^{n} (y_i - \hat{y}_i)^2 \quad \text{subject to } \sum_{j=1}^{p} a_j^2 \leq t
\]
Using AutoBench

> quickBench [functionA, functionB, ...] ["functionAName", "functionBName", ...]

> bench slowRev, [0 ..200], nf

> bench FastRev, [0 ..200], whnf
Assignment

Problem 1, *Comparing Sorting Performance with QuickBench*: Implement quickSort, mergeSort, bubbleSort, and insertionSort. Next, write the QuickCheck test suite by comparing your functions to Haskell’s default “sort” function. Then, make the functions *Benchmarkable* and provide graphical and statistical output comparing the performance of the various sorting methods. QuickCheck and Criterion results must be implemented using the quickBench function.

Problem 2, *Comparing Sorting Performance with AutoBench*: Now, provide a more sophisticated analysis by generating correctness and performance metrics using the autoBench function. Before running the experiments, explain which algorithm you think will be best on small, medium, and large lists. Then after generating performance metrics with autoBench, use statistics from the report to demonstrate which algorithm is best suited for each size input.

Problem 3, *impact of nf vs genNf*: Investigate the impact of generating QuickCheck inputs on time performance. Compare the difference of using *nf* and *genNf* on the sorting algorithms. This analysis must be implemented via the autoBench framework.
Assignment

Problem 4, *Extension of iSets*: Assuming the class has completed the iSet library from HW7, a natural extension involves comparing the time performance of the native haskell Set class with our custom iSet class. Compare the correctness and time performance of the following functions. The tests must use autoBench to complete the analysis.

- Size
- Member
- notMember
- isSubsetOf
- isProperSubsetOf
- Union
- Unions
- Intersection
- Difference
- Filter
- Split
- SplitMember
- maxView
- minView
- fromList
Questions
The Problem

- Purely functional representations of graphs are hard
- Partial Functions
  - Some inputs result in valid graphs, some don’t
- For example:
  - data G a = G { vertices :: [a], edges :: [(a,a)]}
  - Invariant: $E \subseteq V \times V$
    - How do you check this with typing? (not possible)
  - G [1,2,3] [(1,2), (2,3)] ✓
  - G [1] [(1,2)] ×
- Other offenders:
  - Containers library:
    - Adjacency list representation, “index out of range”
  - fgl library:
    - Inductive graph representation, “edge from non-existent vertex”
The Solution

- Abstract away low level details of graph to a higher level, safer, interface where graphs are defined by a set of axioms, similar to how numbers are characterised by rings

Algebraic Graphs
Mokhov’s Algebraic Graphs

data Graph a = Empty
    | Vertex a
    | Overlay (Graph a) (Graph a)
    | Connect (Graph a) (Graph a)

This data structure created by Mokhov forms the basis of his algebraic graphs, and the definition does not allow for partial definitions, as anything that could be represented by this structure is a valid graph, and only valid graphs can be represented by this structure.

Vertex represents a single node, holding a single value a, and Empty is the lack of nodes / the empty graph.

Overlay and Connect are defined using binary operations that mokhov defines as such:

- **Overlay:** $(V_1, E_1) + (V_2, E_2) = (V_1 \cup V_2, E_1 \cup E_2)$
- **Connect:** $(V_1, E_1) \rightarrow (V_2, E_2) = (V_1 \cup V_2, E_1 \cup E_2 \cup V_1 \times V_2)$
Examples

1 \rightarrow 2 \text{ represents the graph containing the nodes } 1 \text{ and } 2, \text{ with an edge connecting them}

1 \rightarrow 2 + 2 \rightarrow 3 \text{ represents the path graph of nodes } \{1,2,3\}

1 \rightarrow (2 + 3) \text{ represents the graph } G (\{1,2,3\}, \{(1,2), (1,3)\})
How this affects Haskell Graphs

Current Graph implementations in Haskell Libraries:

- Can break invariant
- Can’t be statically checked
- Calls for partial functions
- Creation/Insertion in O(N) time

Graphs using Mokhov’s algebraic graphs library:

- Can’t break invariant
- Can be checked both statically and dynamically
- No partial functions
- Creation / Insertion O(1) time
Assignment - Quick Check Module

Purpose: verify the algebraic properties of the algebraic graphs implementation

- + is commutative
  - \( x + y = y + x \)
- + is associative:
  - \( x + (y + z) = (x + y) + z \)
- \((G, \rightarrow, \varepsilon)\) is a monoid:
  - identity: \( \varepsilon \rightarrow x = x, x \rightarrow \varepsilon = x \)
  - associativity: \( x \rightarrow (y \rightarrow z) = (x \rightarrow y) \rightarrow z \)
- \(\rightarrow\) of \(\rightarrow\) over +:
  - \( x \rightarrow (y+z) = x \rightarrow y + x \rightarrow z \)
  - \( (x+y) \rightarrow z = x \rightarrow z + y \rightarrow z \)
- Decomposition of \(\rightarrow\):
  - \( x \rightarrow y \rightarrow z = x \rightarrow y + x \rightarrow z + y \rightarrow z \).
Assignment - Algorithms Module

Purpose: get experience working with algebraic graphs

Implementations:

- Determine if graph is Eulerian
  - Directed Algebraic graphs (Relation Data Type)
  - Undirected (Symmetric Data Type)
- Speed comparison
  - Algebraic graphs are faster at certain operations and slower at others
  - Helps to reinforce this concept
Questions...?
Presentation Overview

KEY TOPICS

Problem / Motivation
Solution
Accelerate
Contributions
Code Example
Proposed Assignment
Conclusion
Questions
Array programs that contain data parallelism are limited by hardware.
Data parallel programs require a certain amount of main memory.
How can we improve efficiency of array-based parallel programs?
Solution

- Give control of sequentialization to the programmer
- Provide a set of sequence combinators
- Use programming language Accelerate as a starting point
- Provide user with ability to specify sequential data-dependency over data-parallel computations
- Provide user with scheduling and streaming functionality
Accelerate

A domain specific functional language for high-performance, multi-dimensional array computations

dotp :: Acc (Vector Float) → Acc (Vector Float) → Acc (Scalar Float)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
Why Use Accelerate?

- Provides functionality for fusions
- Provides framework for handling large data sets
Contributions

- Sequence combinators
- Examples using their combinators
- Functions that convert from arrays to sequences and vice versa
- Functions for mapping over sequences
- Function for folding over sequences
- Functions that convert from sequences to Haskell lists
Sequence Combinators

Example program

dotpSeq :: Acc (Vector Float)
    → Acc (Vector Float)
    → Acc (Scalar Float)

dotpSeq xs ys =
    collect
        $ foldSeqE (+) 0
        $ zipWithSeqE (*) (toSeqE xs) (toSeqE ys)
Proposed Assignment

QUICK SELECT
Implement in standard Haskell and then using sequence combinators

COMPARE
Use GHC.Stats to compare implementations

REPORT
Summarize findings in a paragraph
Conclusion

- Paper was well organized and written
- Sufficiently demonstrated the usefulness of their solution
- Assignment will be useful for students to see the benefits of parallelization
Bridging the GUI Gap with Reactive Values and Relations

By: Ivan Perez and Henrik Nilsson

Eli Bendinsky, AJ Nagashima
Research Paper Summary

- Prior to this paper, there were two different approaches to Graphical User Interfaces in Functional Programming.
- There were several disadvantages to each approach, each very hard to work around.
- The purpose of this paper is to introduce Reactive Values and Relations as a method to bridge both approaches, thus eliminating the issues with each approach.
Previous Approaches

- **Functional Calls to Imperative APIs**
  - Code is inherently not functional like
  - Model-View-Controller
    - Lot of code for this pattern resides in the controller, and expands exponentially

- **Defining a Functional GUI but evaluating the Model with nonfunctional helpers**
  - Widgets are required
    - Must be evaluated using a single input and a single output
    - A widget must be created for every use case

- **Functional Reactive Programming where the state takes the form of a signal**
  - Signals are reactive values based on time and input/output
    - This causes mutual dependency, so an input and output must be defined at the same time as the signal itself
The Bridge

- **Reactive Values**
  - Bind to a value/device to be modified or read by an external stimulus
  - Values can be defined as readable and/or writeable

- **Reactive Relations**
  - Relates two values using a concept called lifting
  - A lift relates one readable relation to a writeable relation
  - A lift can also be multidirectional
How It Works

● An internal value can be any haskell variable type
● An external value can be an input or an output
● Both can be assigned to a Reactive Value
  ○ An input or output and an internal value can then be related through a lift
● A Reactive Relation is assigned to two (or more) Reactive Values
  ○ An input can be written to and read, and when read the input value can be converted to an internal value
The Point: How Does It Bridge The Gap?

- Provides a functional approach to GUIs
  - Code is functional, does not mimic imperative counterparts
- An input and output can be linked in multiple ways
  - A widget’s values can be linked without definitions for each use case
  - Or overcomplicating the codebase
- A relation is defined after a value
  - Linking two values of a relation is defined after declaration, so they are not mutually dependent
The Project

- Often the first step when learning imperative frameworks
- Contains scalable complexity to match a programmer’s skillset
  - Core features include: display of list, ability to create and delete items, view of completed or active items
  - Advanced features can range from assigning due dates to creating tags for tasks
- Provides programmer with context for Reactive elements when compared to a standard MVC approach
- Teaches programmers how to define relations between the display of items and their values
Questions?
"Oh my God, why did you scotch-tape a bunch of hammers together?"

"It's ok! Nothing depends on this wall being destroyed efficiently."

https://xkcd.com/1926/
Research Paper Presentations

- A meta-EDSL for Distributed Web Applications (Haskell’17)
  Fahad Atif and Henry Farr
- Proving Type Class Laws for Haskell (TFP’19)
  Maheen Riaz Contractor and James Coombs
- Backpack: Retrofitting Haskell with Interfaces (POPL’14)
  Jonathan Patten and Adam Spindler
- There is No Fork: An Abstraction for Efficient, Concurrent, and Concise Data Access (ICFP’14)
  Michael Brice and Daniel Londono Osvath
- Pattern Synonyms (ICFP’16)
  Sergey Goldobin and Josh Spangler
Summary of “A Meta-EDSL for Distributed Web Applications” by Anton Ekblad

Fahad Atif and Henry Farr

Motivations

- In the paper, Anton Ekblad presents an embedded domain specific language (EDSL) for building distributed web applications.

- Shortcomings of traditional approaches to building distributed web applications:
  - They consist of multiple independent programs that may be written in different languages.
  - Developers must write ad hoc protocols to support communications between the independent programs.
  - The programs may not be type-safe so a change in one program could negatively impact another.
  - A lot of overhead is present in the network design and communication protocols that connect the individual programs.
The Language

- The language is embedded in Haskell.
- Applications consist of several nodes.
- Pure code can be shared amongst nodes and polymorphic code can be called from any node.
- The language provides type-safe and boilerplate free communication between nodes.
- A distributed web application can be written as a single program that can target various platforms (e.g. on the client-side, the Haskell code could be compiled to JavaScript code for the browser).
Hello World

- This application consists of both server and client nodes.
- Conventional approaches may require:
  - Separate client and server programs.
  - Client-server communication code and protocols.
- Type-safety between client and server nodes.

```hs
1 instance Node Server where
2   endpoint _ = remoteNode "example.com" 24601
3
4 greet :: RemotePtr (String -> Server ()
5 greet = static (remote $ liftIO . putStrLn)
6
7 main = runApp [start (Proxy :: Proxy Server)] $ do
8   dispatch greet "Hello, server!"
```

Figure 2. Hello, server!
Security

- The language provides strict boundaries between nodes.
- Web applications typically have many trust and security related challenges.
  - Client-side code can be modified by the client.
  - Third party dependencies could change and become malicious.
- It is important to treat clients and third party dependencies as adversaries.
- In this language, sandboxed iframes can be integrated into applications as just another node.
- Sandboxed iframes allow code to be executed within them without allowing the code to view or change other parts of the application.
- This is not done often in conventional web application development since it requires message passing support and third-party libraries may not support this.
Sandboxing

- The code for displaying ads from a third-party service is running in a sandbox so it cannot interfere with other parts of the application.

```hs
data AdRotation
  type MySbx = Sandbox AdRotation

instance Node MySbx where
  endpoint = localNode
  type Allowed MySbx a = a -> Client
  init = dependOn "http://example.com/ads.js"

randomAd :: RemotePtr (MySbx URL)
randomAd = static (remote $ liftIO . ffi "getAdvert")

main = runApp [start (Proxy :: Proxy MySbx), ...]
  ...
  setTimer (Repeat (10*60*1000)) $ do
    adURL <- dispatch randomAd
    withElem "ad_banner" $ \
        \banner_img -> do
      setProp banner_img "src" adURL
```

Figure 6. Adding sandboxed ads to the chat client
Flexibility

- Can easily restructure the application and share/move code between nodes.
- The code is written in one language so it is easy to transfer code from one part of the application to another.
- In contrast, the individual programs that make up traditional distributed web applications may be written in different languages so they would need to be translated.
  - This forces developers to make early or premature design decisions to prevent the need for moving functionality.
- Third party EDSLs can easily be integrated into an application since they can be treated as just another node.
Advancing Haskell

- Provides path to integrate different components in applications
- Popularity of cloud computing demands a different approach to writing code
  - One machine, one program
  - Versus many different machines running different components
- Encourages Haskell programmers to think about distributed programming
- Appeals to practical applications (as opposed to research applications) of Haskell
Example

An example from the paper was a photo browsing website, with a built in chat server.

Figure 3. Architecture of our example application
Proposed assignment: Criteria

- Basic web application for a blog-style website
- Features include publishing blog posts, leaving comments, deliberately open ended
- Must handle requests, provide data persistence, basic authentication
- In short, a collection of heterogeneous components
Proposed assignment: Details

- A student would use the EDSL to tie components together
- May be limited in scope
  - Lose appreciation of distributed programing if every node runs on the same machine
  - Could modify it to run an instructor node that student code must use
Proposed assignment: Outcomes

- Showcase Haskell’s adaptability
- Expose students to working with diverse systems
- Force students to acknowledge difficulties of distributed programming
Find out more

- Haste, Haskell-to-JavaScript maintained by the author
- https://haste-lang.org/
- Code is compiled as many times for as many nodes
Questions
Proving Type Classes for Haskell

Presented by:
James Coombs,
Maheen Riaz Contractor
Proving Type Classes for Haskell


Introduction

- Type Classes
  - Ad-hoc polymorphism
  - Invariants
- Haskell does not support verification of Type Classes
- The authors attempt to solve this problem

```
1  Functor : fmap id == id
2    Monad : m >>= return == m
3     Num  : abs x * signum x == x
```
Motivation

- Haskell is all about proving logical relationships and laws
- QuickCheck offers counterexamples
  - Not sufficient for a proof
- The authors attempted to make a prover
Contributions

- Type Class Laws went from:
  - Abstract declarations -> concrete instantiations
  - Informal description   -> enforced documentation
- Created an automatic tool to use induction on Haskell source code
Process

1st Haskell Source
2nd TIP Language
3rd HipSec
4th Standard Out
“In the future, we envisage programmers writing similar properties and laws, and using a tool similar to what’s described here to not only test, but also prove their programs correct, with little more overhead than QuickCheck”
Use Cases

- Replace QuickCheck
- Prove type class laws for first order functions
- Output the total time it took for the proof

Major Limitation:
- Does **NOT** support Higher Order Functions
- Disqualifies Monoids, Functors and Monads
Assignment

- Goals: Understand why...
  - Type Class laws should be proven
  - QuickCheck isn’t sufficient
  - There is a desire for true proof

- Assignment:
  - Write a Num type class implementation
  - Write QuickCheck tests
  - Explain why QuickCheck is insufficient for proof
  - Theorize ways to prove correct implementation
Conclusion

- Good idea
  - Proof is fundamental to logical execution
- Bad execution (so far)
  - Higher Order Functions are fundamental to functional programming
Questions?
Backpack: Retrofitting Haskell with Interfaces

Scott Kilpatrick, Derek Dreyer, Simon Peyton Jones, and Simon Marlow

Presented By
Jonathan Patten & Adam Spindler

Kilpatrick et al. [2014]
Problems with Haskell’s Type Classes

- Haskell’s type classes aren’t very flexible
  - When making a type an instance of a typeclass, must provide implementations for all functions

- Haskell only allows a single implementation of a type class for a given data type
  - For example, Monoid for the Num type has two obvious implementations

- Haskell started off as a research language, may be why this wasn’t built in
newtype Sum a = Sum { getSum :: a }
newtype Product a = Product { getProduct :: a }

instance Num a => Monoid (Sum a) where
    mappend (Sum a) (Sum b) = Sum $ a + b
    mempty = Sum 0

instance Num a => Monoid (Product a) where
    mappend (Product a) (Product b) = Product $ a * b
    mempty = Product 1
Why do we need Interfaces?

- Make code more generic
  - Could be multiple implementations of String (both UTF-8 and UTF-16)
- Allow for mocking of objects for unit tests
- Could implement with a data type and pass instances around

```haskell
data NumMonoid = NumMonoid {
  mappend :: Num a => a -> a -> a,
  mempty :: Num a => a
}
```
Why use Backpack?

- The basic idea behind Backpack is to late-bind modules in the package manager.
- The goal is to limit code duplication and allow the compiler to in-line functions by type-checking before the implementation is known.
- Ideally, importing a module using Backpack should be no more expensive than importing a module normally.
How does it work?

- Abstract module bindings, called holes
- Holes require signatures, which define the function types required by the module
- Example signature allowing for different implementations of strings:

```haskell
signature Str where
  data Str
  append :: Str -> Str -> Str
  Index :: Str -> Char -> Int
  etc.
```
Three situations where modules share names:

Hole-Hole: Backpack will merge the implementations together, effectively joining their signatures

Mod-Hole: The module must be a sub-type of the hole, it “fills” the hole if it defines all the same entities declared in the hole

Mod-Mod: Only succeeds if the modules share the same implementation

Ex: Packages left and right import package top, package bottom imports left and right, the result is well typed because left and right provide the same module top
Using BackPack

- Both .b kp files for testing, and production support in Cabal
- Can run .b kp files with ghci --backpack my_backpack_test.b kp
```
unit test where
  signature Chr where
data Chr
  upper :: Chr -> Chr

module StringOps where
  import Chr

  upperCase :: [Chr] -> [Chr]
  upperCase = map upper

module Main where
  import qualified StringOps.Char

  main = print (StringOps.Char.upperCase ['a'])
```
There is no Fork: an Abstraction for Efficient Concurrent and Concise Data Access

Simon Marlow, Louis Brandy, Jonathan Coens, Jon Purdy

Presented By: Daniel Osvath, Michael Brice
Motivation
Accessing External Data

Web-based applications

Multiple external data sources

Efficiency in fetching requires concurrency

If read order does not matter -> Concurrency can be managed automatically
Abstract Concurrency

Avoid typical concurrency overhead (fork, join threads)

Let the programmer focus on business logic

Keep business logic concise, readable

Enable efficient and modular code
Solution
Applicative Abstraction

Concurrency is implicit

Use Applicative Combinators <$> and <*>

Use cache to memoize results of previous data fetches

Takes a set of functions that do a unit of work that involves and external source, packaging each function into a Blocked, Done container.

A main loop will iterate synchronously through all the containers returning what is done, and attempting to invoke anything that is blocked.
Example - Rendering a Blog

renderPage :: Html -> Html -> Html

Blog :: Fetch Html
Blog = renderPage <$> leftPane <*> mainPage

leftPane :: Fetch Html
leftPage = renderSidePane <$> popularPosts <*> topics
Example - Rendering a Blog

getPostDetails :: PostId
   -> Fetch (PostInfo, PostContent)

getPostDetails pid =
   (,) <$> getPostInfo pid <*> getContent pid
Example

Regular
length (intersection (fromList A) (fromList B))

Concurrent
length <$> intersection (fromList A) (fromList B)

Where A and B are some expensive call to get a list
Contribution
Real world usage

Proven to be effective - used with Facebook.

Easy to work with thanks to applicative style.

Maximize concurrency without the sacrifice to conciseness.

Automated concurrency.
Helps haskell compete

Ease of use - makes it easy for newcomers to the language to benefit.

Speedup helps it keep pace with most popular OO languages.

Minimal understanding needed to use - advantage other languages lack.
Assignment
Assignment - Fetch Conference Data

WikiCFP website

Concurrently fetch the first 5 pages

(Concurrently) Parse the events on the page

Pretty print the list of events

wikicfp.com/cfp/call?conference=computer%20science&page=1
Assignment - Fetch Conference Data

Haskell Parser Library Provided

(or builds on Parser completed in previous assignment)

Custom table print function

<table>
<thead>
<tr>
<th>name</th>
<th>start_date</th>
<th>end_date</th>
<th>location</th>
<th>deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE---ICoIAS---Ei and Scopus 2020</td>
<td>2020-02-26</td>
<td>2020-02-29</td>
<td>Singapore</td>
<td>2019-11-20</td>
</tr>
<tr>
<td>CCWC 2020</td>
<td>2020-01-06</td>
<td>2020-01-08</td>
<td>Las Vegas, NV, USA</td>
<td>2019-11-20</td>
</tr>
<tr>
<td>ACM-ACAI---Ei/Scopus 2019</td>
<td>2019-12-20</td>
<td>2019-12-22</td>
<td>Sanya, China</td>
<td>2019-11-20</td>
</tr>
<tr>
<td>ACM-MLNLP-Ei/Scopus 2019</td>
<td>2019-12-20</td>
<td>2019-12-22</td>
<td>Sanya, China</td>
<td>2019-11-20</td>
</tr>
<tr>
<td>IFAC World Congress SIMCA 2020</td>
<td>2020-07-12</td>
<td>2020-07-17</td>
<td>Berlin, Germany</td>
<td>2019-11-20</td>
</tr>
<tr>
<td>ICIIC---Ei, Scopus 2020</td>
<td>2020-01-10</td>
<td>2020-01-12</td>
<td>Xiamen, China</td>
<td>2019-11-20</td>
</tr>
<tr>
<td>AECC---Ei and Scopus 2020</td>
<td>2020-01-06</td>
<td>2020-01-08</td>
<td>Rabat, Morocco</td>
<td>2019-11-20</td>
</tr>
<tr>
<td>ACM---ICACS---Ei Compendex and Scopus 2020</td>
<td>2020-01-06</td>
<td>2020-01-08</td>
<td>Rabat, Morocco</td>
<td>2019-11-20</td>
</tr>
</tbody>
</table>
Questions
Agenda

● Background
  ○ Why do we like pattern matching?

● Pattern Synonyms
  ○ Motivation and key idea
  ○ Types of Synonyms:
    ■ Implicit Bidirectional
    ■ Unidirectional
    ■ Explicit Bidirectional

● Contribution
  ○ How does this paper advance Haskell as a whole?
Background and Motivation

- You hate to see that →
- Pattern matching is expressive, but can be cumbersome.
- We want to abstract common portions of multiple patterns.
- Want to make code more clear and readable at a glance.

```haskell
{-
  Matrix data structure that stores its dimensions and the layout of the
  matrix in box row and column major orders.
-}
data Matrix a = Matrix Int Int [[a]] [[a]]

{-
  Get the contents of the first column of the matrix.
-}
firstCol :: Matrix a -> [a]
firstCol (Matrix _ _ _ (col : _)) = col
```
Pattern Synonyms

- Allow for clean and efficient implementations.
- Are a lot clearer to the programmer
- Modular and reusable!
  - We like that a lot.
- Several different kinds
  - Will cover shortly.
- Are fully featured and implemented by the authors of the paper.

```haskell
{-
  Declare a pattern that matches and binds a variable to the first column in a matrix.
-}
pattern ColOne :: [a] -> Matrix a
pattern ColOne col <- (Matrix _ _ _ (col : _))

{-
  Get the contents of the first column of the matrix.
-}
firstCol :: Matrix a -> [a]
firstCol ColOne col = col
```
Implicitly Bidirectional Patterns

- Simplest form
- Can be interpreted both as a pattern, and as an expression.
- Act as pseudo data constructors in both directions, hence the name.
- Can be nested and arbitrarily complex on the right side.
- Declared with the equality operator.

```haskell
{-
  A simple Implicit Bidirectional Pattern
-}
pattern Just2 a b :: Maybe a -> Maybe b -> Maybe (a, b)
pattern Just2 a b = Just (a, b)

{-
  A function utilizing the pattern.
-}
foo :: Maybe (Int, Int) -> Int
foo (Just2 x y) = x + y

{-
  Patterns can be nested within each other!
-}
pattern Complex :: a -> b -> c -> (Maybe ([a], b), String, c)
pattern Complex a b c = (Just2 [a] b, "2", c)
```
Unidirectional Patterns

- Patterns that make sense as a match, but not as a data constructor.
- Can only be interpreted as a pattern
- Was used in the Matrix example
- Declared with a left arrow operator.

```haskell
{-
  This does not make sense!
  What happens if somebody tries:
  foo = Head 3
-}
pattern Head x = x : _  

{-
  This conveys what we want:
  A pattern that matches a non-empty list and binds a variable to its head.
-}
pattern Head :: a -> [a]
pattern Head x <- x : _
```
Explicitly Bidirectional Patterns

- Data structure representing a point in Cartesian Coordinates
- Data Point = CP Float Float
- Convert a point in Cartesian coordinates to Polar.
  pointToPolar :: Point -> (Float, Float)
  pointToPolar (CP x y) = ...
- Convert a point in Polar coordinates to Cartesian.
  polarToPoint :: Float -> Float -> Point
  polarToPoint r a = ...

- Match a Point in Cartesian coordinates and bind the variables for an appropriate Polar representation.
- Pattern Polar :: Float -> Float -> Point
  pattern Polar r a <- (pointToPolar -> (r, a))
  where
    Polar r a = polarToPoint r a
- A function that takes a Point in Cartesian coordinates, performs a computation in terms of polar coordinates, then hands back a new Cartesian Point.
- DoMath :: Point -> Point
  doMath Polar r a = polarToPoint r' a'
  where r' = ...
    a' = ...

- Most powerful and expressive kind.
- Can be interpreted as a pattern, but conversion to expression requires additional explicit work – hence the name.
- Consist of a matcher and a builder
  - Often symmetric, but don’t have to be!
Proposed Assignment - Overview

- Inspired by the game of GTicTacToe from assignment 5
- Broken into 3 main parts:
  1. Initial implementation
  2. Pattern synonym identification
  3. Reimplementation
- Desired purpose: realize how pattern synonyms simplify the use of larger/complicated types (such as game boards)
Proposed Assignment - Implementation Details

- Will be supplied with `mnkGame.hs`
  - Handles all IO
  - Outlines undefined game methods
- Room for two implementations of each game method
  - For parts 1 and 3
- Part 2 is filled out in the comments of the game methods
- Game methods: `generateBoard`, `isWinRow`, `isWinCol`, `isWinDiag`, `isTie`, etc.

```haskell
{-
  Initial Implementation of generateBoard method
-}
generateBoard1 :: Int -> Int -> Board
generateBoard1 = undefined

{-
  Reimplementation of generateBoard
  Identify any types of pattern synonyms that can be used:

  <<Your answer here>>
-}
generateBoard2 :: Int -> Int -> Board
generateBoard2 = undefined
```
Proposed Assignment - Examples

- A game board data structure may be:
  ```haskell
data Board m n k = Board (m, n, k, [[String]])
```

- `isWinRow1` may be implementing using complicated pattern matching:
  ```haskell
isWinRow1 :: Int -> Board -> Bool
isWinRow1 i (Board (_, _, _, (row : rows))) = ...
```

- `isWinRow2` may be implemented more cleanly using pattern synonyms:
  ```haskell
pattern RowView r rs <- Board (_, _, _, (row : rows))
isWinRow2 :: Int -> Board -> Bool
isWinRow2 i (RowView r rs) = ...
```
Contribution & Conclusion

- The paper introduces the notion of Pattern Synonyms
  - The authors fully implemented a pattern synonym library.
    - Modular, flexible, rich typing system, widespread success.
- Improved code clarity and readability, as well as at-a-glance comprehension.
  - As a result, improves overall code quality.
- Haskell as a language is made better by the existence of the Pattern Synonym library.
- Authors’ evaluation revealed a lot of untapped potential.
The End.

Any questions?
The problem with Haskell is that it’s a language built on lazy evaluation and nobody’s actually called for it.

https://xkcd.com/1312/