Haskell I/0 and Pure Computation

COS 441 Slides 5

Slide content credits:
Paul Hudak's Haskell School of Expression

Agenda

- Haskell so far
  - Pure computation
  - Reasoning about programs by substitution of equals for equals
- This time:
  - I/O

SUBSTITUTION OF EQUALS FOR EQUALS

Substitution of Equals for Equals

- A key law about Haskell programs:

let x = <exp> in
... x ... x ...

=

... <exp> ... <exp> ...

- For example:

let x = 4 `div` 2 in
x + 5 + x

= (4 `div` 2) + 5 + (4 `div` 2)

= 9

Substitution of Equals for Equals

- We'd also like to use functional abstraction without penalty

```
halve :: Int -> Int
halve n = n `div` 2
```

- And instead of telling clients about all implementation details, simply expose key laws:

Lemma 1: for all n, if n is even then (halve n + halve n) = n

- Now we can reason locally within the client:

let x = halve 4 in x + x	=	(halve 4) + 5 + (halve 4)	(substitution)
	=	(halve 4) + (halve 4) + 5	(arithmetic)
	=	4 + 5	(Lemma 1)
	=	9	(arithmetic)

Computational Effects

- What happens when we add mutable data structures?
- Consider this C program:

```
int x = 0;

int foo (int arg) {
  x = x + 1;
  return arg + x;
}
```

- We lose a lot of reasoning power!

```
int y = foo (3);
int z = y + y;
```

≠

```
int z = foo (3) + foo (3);
```

Computational Effects

- What happens when we add mutable data structures?
- Consider this C program:

```
int x = 0;

int foo (int arg) {
  x = x + 1;
  return arg + x;
}
```

- We lose a lot of reasoning power!

```
int y = foo (3);
int z = y + y;
```

≠

```
int z = foo (3) + foo (3);
```

↓ 8 ↓ 9

Computational Effects

- What happens about I/O?

```
int foo (int arg) {
  printInt arg
  return arg;
}
```

- We lose a lot of reasoning power!

```
int y = foo (3);
int z = y + y;
```

↓

6 printing "3"

$\neq$

```
int z = foo (3) + foo (3);
```

↓

6 printing "33"

Computational Effects

- A function has an effect if its behavior cannot be specified exclusively as a relation between its input and its output
  - I/O is an effect
  - An update of a data structure is an effect
- When functions can no longer be described exclusively in terms of the relationship between arguments and results
  - many, many fewer equational laws hold:

let x = <exp> in ... x ... x ... ≠ ... <exp> ... <exp> ...

- Rats! What does Haskell do?
  - we need effects like reading and writing files, displaying graphics, playing music, etc...
  - we want equational reasoning

HASKELL EFFECTS
INPUT AND OUTPUT

I/O in Haskell

- Haskell has a special kind of value called an action that *describes* an effect on the world

- Pure actions, which just do something and have no interesting result are values of type IO ()

- Eg: putStr takes a string and yields an action describing the act of displaying this string on stdout

```
-- writes string to stdout
putStr :: String -> IO ()

-- writes string to stdout followed by newline
putStrLn :: String -> IO ()
```

I/O in Haskell

- When do actions actually happen?

- Actions happen under two circumstances:*
  1. the action defined by main happens when your program is executed
     - ie: you compile your program using ghc; then you execute the resulting binary
  2. the action defined by any expression happens when that expression is written at the ghci prompt

* there is one other circumstance: Haskell contains some special, unsafe functions that will perform I/O, most notably System.IO.Unsafe.unsafePerformIO

I/O in Haskell

hello.hs:

```
main :: IO ()
main = putStrLn "Hello world"
```

in my shell:

```
dpw@schenn ~/cos441/code/Trial
$ ghc hello.hs
[1 of 1] Compiling Main             ( hello.hs, hello.o )
Linking hello.exe ...

dpw@schenn ~/cos441/code/Trial
$ ./hello.exe
hello world!
```

bar.hs:

```
bar :: Int -> IO ()
bar n =
  putStrLn (show n ++ " is a super number")

main :: IO ()
main = bar 6
```

in my shell:

```
dpw@schenn ~/cos441/code/Trial
$ ghcii.sh
GHCi, version 7.0.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude> :l bar
[1 of 1] Compiling Main             ( bar.hs, interpreted )
Ok, modules loaded: Main.
*Main> bar 17
17 is a super number
*Main> main
6 is a super number
*Main>
```

Actions

- Actions are descriptions of effects on the world. Simply writing an action does not, by itself cause anything to happen

bar.hs:

```
hellos :: [IO ()]
hellos = [putStrLn “Hi”,
          putStrLn “Hey”,
          putStrLn “Top of the morning to you”]

main = hellos !! 2
```

in my shell:

```
Prelude> :l hellos
...
*Main> main
Top of the morning to you
*Main>
```

Actions

- Actions are just like any other value -- we can store them, pass them to functions, rearrange them, etc:

```
sequence_ :: [IO ()] -> IO ()
```

baz.hs:

```
hellos :: [IO ()]
hellos = [putStrLn "Hi",
          putStrLn "Hey",
          putStrLn "Top of the morning to you"]

main = sequence_ (reverse hellos)
```

in my shell:

```
Prelude> :l hellos
...
*Main> main
Top of the morning to you
Hey
HI
```

Combining Actions

- The infix operator >> takes two actions a and b and yields an action that describes the effect of executing a then executing b afterward

```
howdy :: IO ()
howdy = putStr “how” >> putStrLn “dy”
```

- To combine many actions, use do notation:

```
bonjour :: IO ()
bonjour = do putStr “Bonjour!”
             putStr “ ”
             putStrLn “Comment ca va?”
```

Combining Actions

- The infix operator >> takes two actions a and b and yields an action that describes the effect of executing a then executing b afterward

```
howdy :: IO ()
howdy = putStr "how" >> putStrLn "dy"
```

- To combine many actions, use do notation:

```
bonjour :: IO ()
bonjour = do putStr "Bonjour!"
             putStr "  "
             putStrLn "Comment ca va?"
```

layout: first non-space after do defines indentation level

Combining Actions

- The infix operator >> takes two actions a and b and yields an action that describes the effect of executing a then executing b afterward

```
howdy :: IO ()
howdy = putStr “how” >> putStrLn “dy”
```

- To combine many actions, use do notation:

```
bonjour :: IO ()
bonjour = do
  putStrLn “Bonjour!”
  putStrLn “”
  putStrLn “Comment ca va?”
```

layout: first non-space after do defines indentation level

Quick Aside: Back to SEQEQ*

- Do we still have it? Yes!

```
let a = PutStrLn "hello" in
do
   a
   a
```

=

```
do
   PutStrLn "hello"
   PutStrLn "hello"
```

* SEQEQ = substitution of equals for equals

Input Actions

- Some actions have an effect and yield a result:

```
-- get a line of input
getLine :: IO String

-- get all of standard input until end-of-file encountered
getContents :: IO String

-- get command line argument list
getArgs :: IO [String]
```

- What can we do with these kinds of actions?
  - we can extract the value and sequence the effect with another:

Input Actions

- Some actions have an effect and yield a result:

```
-- get a line of input
getLine :: IO String

-- get all of standard input until end-of-file encountered
getContents :: IO String

-- get command line argument list
getArgs :: IO [String]
```

- What can we do with these kinds of actions?
  - we can extract the value and sequence the effect with another:

```
do
  s <- getLine
  putStrLn s
```

Input Actions

- Some actions have an effect and yield a result:

```
-- get a line of input
getLine :: IO String

-- get all of standard input until end-of-file encountered
getContents :: IO String

-- get command line argument list
getArgs :: IO [String]
```

- What can we do with these kinds of actions?
  - we can extract the value and sequence the effect with another:

```
do
  s <- getLine
  putStrLn s
```

s has type string

getLine has type IO string

Input Actions

- A whole program:

```
main :: IO ()
main = do
    putStrLn “What’s your name?”
    s <- getLine
    putStr “Hey, “
    putStr s
    putStrLn “, cool name!”
```

import modules

contains readFile

contains getArgs, getProgName

```
import System.IO
import System.Environment

processArgs :: [String] -> String
processArgs [a] = a
processArgs _   = ""

echo :: String -> IO ()
echo "" = putStrLn "Bad Args!"
echo fileName = do
  s <- readFile fileName
  putStrLn "Here it is:"
  putStrLn "***********"
  putStr s
  putStrLn "\n***********"

main :: IO ()
main = do
  args <- getArgs
  let fileName = processArgs args
  echo fileName
```

<- notation:
RHS has type IO T
LHS has type T

let notation:
RHS has type T
LHS has type T

SEQEQ (Again!)

- Recall: s1 ++ s2 concatenates String s1 with String s2
- A valid reasoning step:

```
let s = "hello" in
 do
   putStrLn (s ++ s)
```

=

```
do
  putStrLn ("hello" ++ "hello")
```

SEQEQ (Again!)

- Recall: s1 ++ s2 concatenates String s1 with String s2
- A valid reasoning step:

```
let s = "hello" in
 do
   putStrLn (s ++ s)
```

=

```
do
   putStrLn ("hello" ++ "hello")
```

- A valid reasoning step:

```
do
   let s = "hello"
   putStrLn (s ++ s)
```

=

```
do
   putStrLn ("hello" ++ "hello")
```

SEQEQ (Again!)

- Recall: s1 ++ s2 concatenates String s1 with String s2
- A valid reasoning step:

```
let s = "hello" in
 do
   putStrLn (s ++ s)
```

=

```
do
  putStrLn ("hello" ++ "hello")
```

- A valid reasoning step:

```
do
  let s = "hello"
  putStrLn (s ++ s)
```

=

```
do
  putStrLn ("hello" ++ "hello")
```

- Wait, what about this:

```
do
  s <- getLine
  putStrLn (s ++ s)
```

≠

```
do
  putStrLn (getLine ++ getLine)
```

wrong type:
getLine :: IO String

SEQEQ (Again!)

- Invalid reasoning step?

```
let s = getLine in
do
   putStrLn (s ++ s)
```

$\stackrel{?}{=}$

```
do
  putStrLn (getLine ++ getLine)
```

SEQEQ (Again!)

- Invalid reasoning step?

```
let s = getLine in
do
   putStrLn (s ++ s)
```

$\overset{?}{=}$

```
do
  putStrLn (getLine ++ getLine)
```

wrong type:
s :: IO String

wrong type:
getLine :: IO String

SEQEQ (Again!)

- Invalid reasoning step?

```
let s = getLine in
do
   putStrLn (s ++ s)
```

?
=

```
do
  putStrLn (getLine ++ getLine)
```

wrong type:
s :: IO String

wrong type:
getLine :: IO String

- The Haskell type system shows x <- e is different from let x = e
  - x has a different type in each case
  - let x = e enables substitution of e for x in what follows
  - x <- e does not enable substitution -- attempting substitution leaves you with code that won't even type check because x and e have different types (type T vs. type IO T)

The Larger Consequences of SEQEQ

- SEQEQ is a technical, mathematical property of a programming language
- What can we say about it's effect on programmers in real life?
- Personal opinion:
  - there's an initial barrier to entry when it comes to functional programming
    - you have to retrain your brain to think in a different way
    - but if you like computer science and programming, you'll probably find that doing the retraining is pretty fun!
    - we don't have that much time in this class to do a ton of retraining so you'll have to continue on your own
  - once you get past the hump, for many applications, it's really is a lot easier to write programs quickly, correctly and concisely
  - SEQEQ, coupled with a strong type system, is a part of that

SEQEQ & Other Languages

- Haskell has full-blown SEQEQ
- C, Java, Python have none
  - functions usually have effects
  - functions usually update object state to get their job done
  - you usually can't reason like you do in Haskell
- Other functional languages like SML, O'Caml, F# go half way
  - data structures are immutable by default (you have to work a little harder to get mutable data structures)
  - functions usually do not have effects
  - functions can usually be specified entirely by a relation between their arguments and their results
  - you can often reason like you do in Haskell
  - I like these other languages a lot -- it's the immutable data structures (and the types) that make 90% of the difference

GRAPHICS

Graphics Preliminaries

```
type Title  = String
type Size   = (Int, Int)
type Point = (Int, Int)

openWindow    :: Title -> Size -> IO Window
closeWindow   :: Window -> IO ()
drawInWindow :: Window -> Graphic -> IO ()
runGraphics     ::  IO () -> IO ()
text                  :: Point -> String -> Graphic
getKey              :: Window -> IO Char
```

the types are descriptive!

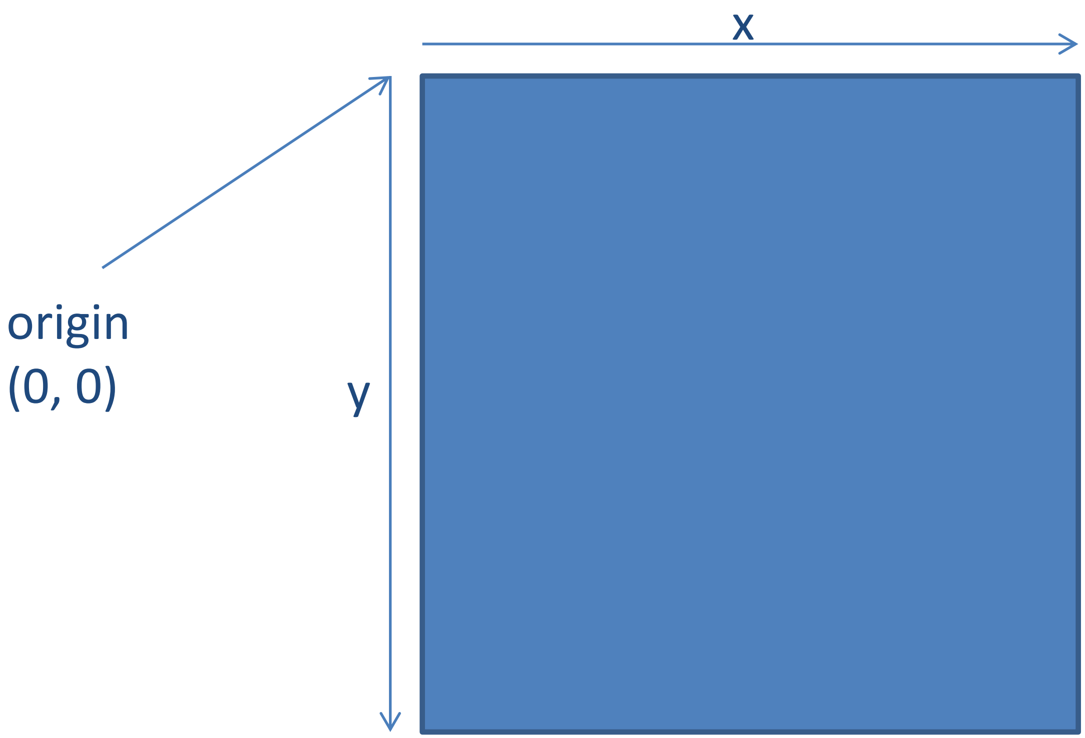
Graphics Preliminaries

```
type Title  = String
type Size   = (Int, Int)
type Point = (Int, Int)

openWindow    :: Title -> Size -> IO Window
closeWindow   :: Window -> IO ()
drawInWindow :: Window -> Graphic -> IO ()
runGraphics     ::  IO () -> IO ()
text                 :: Point -> String -> Graphic
getKey             :: Window -> IO Char
```

- A first program:

```
main =
  runGraphics (
  do w <- openWindow "My prog" (300, 300)
       drawInWindow w (text (10, 20) "Hello World")
       k <- getKey w
       closeWindow w )
```

Graphics Window



Recursive functions & do notation

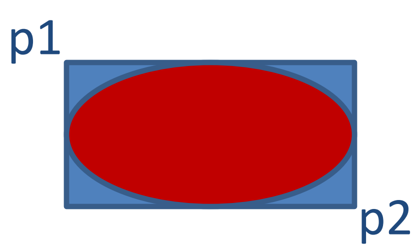
```
spaceClose :: Window -> IO ()

spaceClose w = do
    k <- getKey w
    if k == ' ' then closeWindow w
                else  spaceClose w
```

Recursive functions & do notation

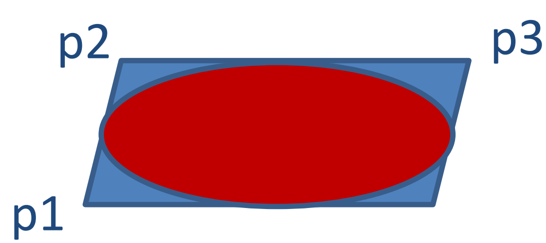
```
spaceClose :: Window -> IO ()

spaceClose w = do
    k <- getKey w
    if k == ' ' then closeWindow w
                else  spaceClose w


main =
  runGraphics (
  do w <- openWindow "My prog" (300, 300)
       drawInWindow w (text (10, 20) "Hello World")
       spaceClose w
  )
```

Other Graphics

```
ellipse         :: Point -> Point -> Graphic
shearEllipse :: Point -> Point -> Point -> Graphic
line              :: Point -> Point -> Graphic
polyline       :: [Point] -> Graphic
polygon        :: [Point] -> Graphic
polyBezier    :: [Point] -> Graphic

withColor     :: Color -> Graphic -> Graphic
data Color = Black | Blue | Green | Cyan | Red
                  | Magenta | Yellow | While
```



ellipse p1 p2

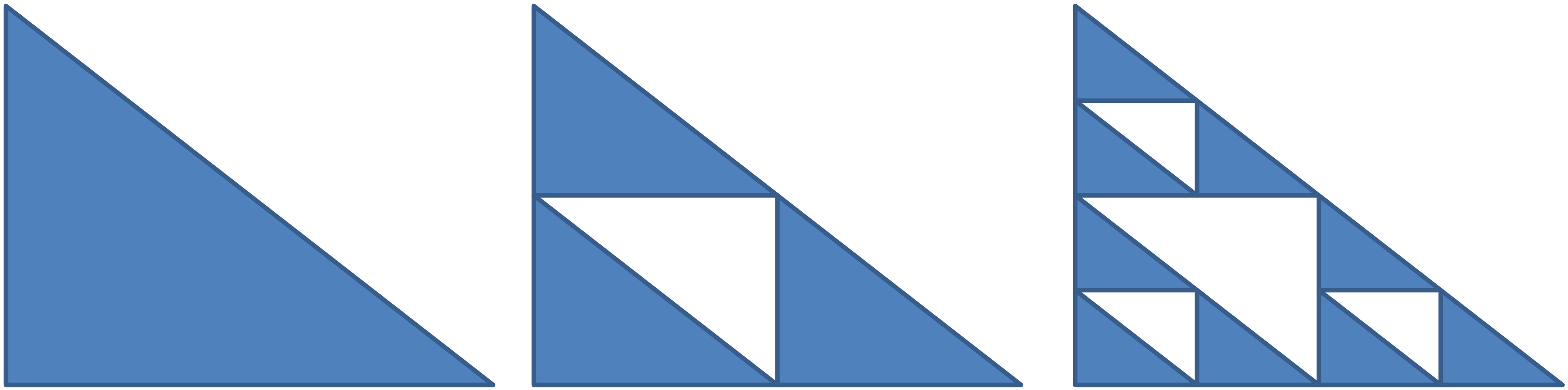


shearEllipse p1 p2 p3

line p1 p2

Fractals

- Fractals are mathematical structures that repeat themselves infinitely often in successively finer detail
- Fractals are often use to simulate natural phenomena: Snow flakes, forests, mountains
- Simple fractals repeat geometric shapes
- Sierpinski's triangle, 3 iterations:

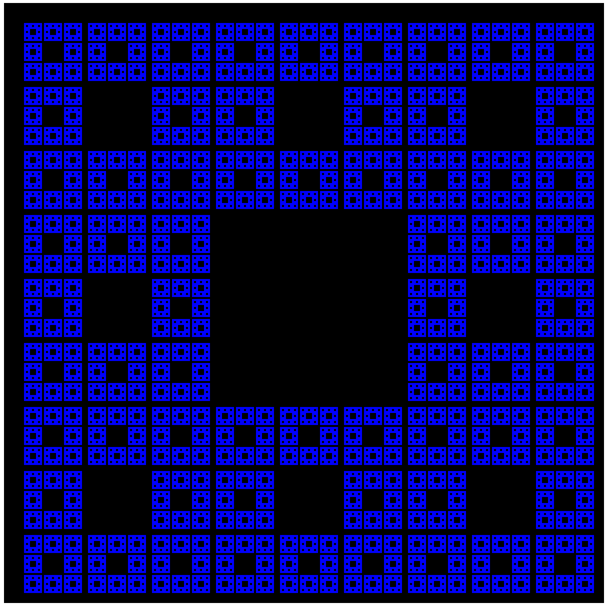


Sierpinski's Triangle

- Let's look at the code ... go to demo

Sierpinski's Carpet

- For your assignment, you'll be constructing Sierpinski's carpet and other fractals:



SUMMARY

Summary

- Haskell I/O
  - actions describe effects
  - do notation sequences actions
  - only the main action (or an action placed at the ghci prompt) is ever executed

- Haskell enjoys referential transparency
  - this powerful reasoning principle allows programmers to substitute definitions for their names whenever they want to
  - C, Java don't have it
  - Other functional languages like F#, O'Caml, SML go half way by making data structures immutable by default
    - In my experience, by limiting effects, these functional languages really do make it easier to write correct code in many domains