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
A key law about Haskell programs:

\[
\begin{align*}
\text{let } x &= \text{exp} \text{ in } \\
... \ x \ ... \ x ... &= ... \ \text{exp} \ ... \ \text{exp} ... \\
\end{align*}
\]

For example:

\[
\begin{align*}
\text{let } x &= 4 \ `\text{div}` 2 \text{ in } \\
x + 5 + x &= (4 `\text{div}` 2) + 5 + (4 `\text{div}` 2) \\
\end{align*}
\]

\[
\begin{align*}
&= 9
\end{align*}
\]
Substitution of Equals for Equals

• We'd also like to use **functional abstraction** without penalty

  \[
  \text{halve :: Int -> Int} \\
  \text{halve n = n `div` 2}
  \]

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

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

• Now we can reason locally within the client:

  \[
  \begin{align*}
  \text{let } x &= \text{halve 4 in } x + x \\
  &= (\text{halve 4}) + 5 + (\text{halve 4}) & \text{(substitution)} \\
  &= (\text{halve 4}) + (\text{halve 4}) + 5 & \text{(arithmetic)} \\
  &= 4 + 5 & \text{(Lemma 1)} \\
  &= 9 & \text{(arithmetic)}
  \end{align*}
  \]
Computational Effects

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

```c
int x = 0;

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

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

• We lose a lot of reasoning power!
Computational Effects

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

```c
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?

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

• We lose a lot of reasoning power!

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

6 printing "3"

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

6 printing "33"
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:

\[
\text{let } x = \langle \text{exp} \rangle \text{ in } ... x ... x ... \neq ... \langle \text{exp} \rangle ... \langle \text{exp} \rangle ...
\]

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:

```haskell
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:

```haskell
bar :: Int -> IO ()
bam 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
Hey
HI
```
Combining Actions

• The infix operator \( \gg \) takes two actions \( a \) and \( b \) and yields an action that describes the effect of executing \( a \) then executing \( b \) afterward.

```haskell
howdy :: IO ()
howdy = putStr “how” \( \gg \) putStrLn “dy”
```

• To combine many actions, use do notation:

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

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

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

- To combine many actions, use do notation:

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

  layout: first non-space after do defines indentation level
Combining Actions

• The infix operator \( \gg \) 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!

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

* SEQEQ = substitution of equals for equals
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
    putStrLn has type IO ()
• A whole program:

```haskell
main :: IO ()
main = do
  putStrLn "What's your name?"
  s <- getLine
  putStrLn "Hey, "
  putStrLn s
  putStrLn ", cool name!"
```
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
SEQEQ (Again!)

• Recall: \( s1 ++ s2 \) concatenates \texttt{String} \( s1 \) with \texttt{String} \( s2 \)

• A valid reasoning step:

\[
\text{let } s = "hello" \text{ in do}
\text{putStrLn (s ++ s)} \quad = \quad \text{do}
\text{putStrLn ("hello" ++ "hello")}
\]
• Recall: \( s_1 ++ s_2 \) concatenates String \( s_1 \) with String \( s_2 \)

• A valid reasoning step:

\[
\text{let } s = "hello" \text{ in } \\
\text{do } \\
\text{putStrLn } (s ++ s) \quad = \quad \text{do } \\
\text{putStrLn } ("hello" ++ "hello")
\]

• A valid reasoning step:

\[
\text{do } \\
\text{let } s = "hello" \\
\text{putStrLn } (s ++ s) \quad = \quad \text{do } \\
\text{putStrLn } ("hello" ++ "hello")
\]
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)
```
• Invalid reasoning step?

```haskell
let s = getLine in
  do
    putStrLn (s ++ s)
```
```haskell
= ?
```
```haskell
do
  putStrLn (getLine ++ getLine)
```
• Invalid reasoning step?

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

wrong type:
s :: IO String

wrong type:
getLine :: IO String
Invalid reasoning step?

The Haskell type system shows \( x <\ -\ e \) is different from \( \text{let } x = e \)

- \( x \) has a different type in each case
- \( \text{let } x = e \) 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 \( \text{IO } T \))

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

==

do
  putStrLn (getLine ++ getLine)
```

wrong type:
\( s :: \text{IO } \text{String} \)

wrong type:
\( \text{getLine} :: \text{IO } \text{String} \)
The Larger Consequences of SEQEQ

- SEQEQ is a technical, mathematical property of a programming language
- What can we say about its 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 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!
type Title = String
type Size = (Int, Int)
type Point = (Int, Int)

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

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

- Fractals are mathematical structures that repeat themselves infinitely often in successively finer detail.
- Fractals are often used to simulate natural phenomena: Snowflakes, 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