Haskell: I/O continued
Haskell: Data

CS 331 Programming Languages

Lecture Slides

Monday, March 6, 2023

Glenn G. Chappell

Department of Computer Science

University of Alaska Fairbanks

ggchappell@alaska.edu

Review

Review
Haskell: I/O — String Conversion [1/2]

A Haskell **typeclass** (or simply **class**) is a collection of types that implement a common interface. Haskell does overloading *only* via typeclasses.

Some commonly used typeclasses:

- `Eq`: **Equality-comparable** types. Overloads: `==`, `/=` (inequality).
- `Ord`: **Orderable** types. Overloads: `<`, `<=`, `>`, `>=`.
- `Num`: **Numeric** types. Overloads: binary `+`, `-`, `*`, `abs`, etc.
- `Show`: **Showable** types. Overloads: `show` (convert to `String`).
- `Read`: **Readable** types. Overloads: `read` (convert from `String`).

The last two typeclasses above involve `String` conversion and so are important when doing I/O.

But note that Haskell keeps `String` conversion and I/O largely separate.

For code from this topic, see `io.hs`.

Review
Haskell: I/O — String Conversion [2/2]

To convert a value of a showable type to a `String`, pass it to `show`.

```
> show [True, False]
"[True,False]"
```

To convert a `String` to a readable type, pass the `String` to `read`. Type annotations are sometimes needed.

```
> fivestr = "5"
> 2 + read fivestr
7
> read fivestr  -- Convert String to ... what?
```

[*Error*]

```
> (read fivestr)::Integer
5
```

Review
Haskell: I/O — Simple Output, About I/O Actions [1/4]

I/O would seem to involve side effects—which Haskell forbids.

A side effect *description* is stored in a Haskell **I/O action**.

Example. Function `putStr` takes a `String` and returns an I/O action representing printing the `String` to the standard output.

Return type: I/O action wrapping a “nothing” value

```
> :t putStr
putStr :: String -> IO ()
```

In a Haskell program, the return value of `main` should be an I/O action. The side effects described by its return value are performed by the runtime environment.

Here is a Haskell hello-world program.

```
main = putStrLn "Hello, world!"
```

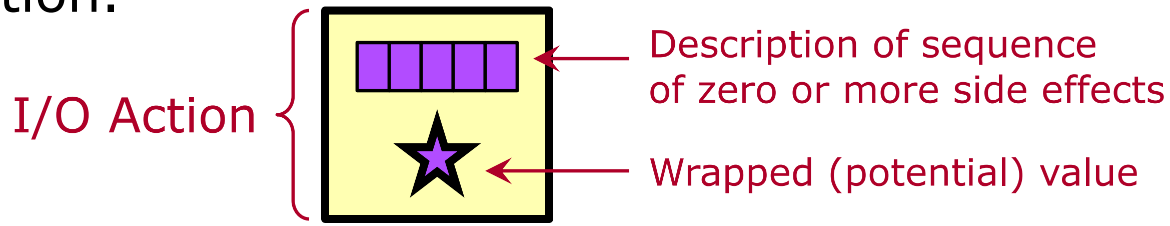
Like `putStr`, but add a newline at the end of the given `String`.

Review
Haskell: I/O — Simple Output, About I/O Actions [2/4]

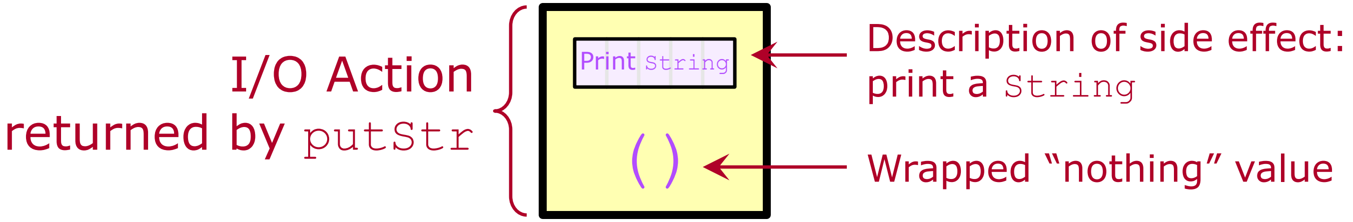
An I/O action includes:

- a description of a sequence of zero or more side effects, and
- a wrapped (potential) value.

Here is an illustration.


I/O Action
Description of sequence of zero or more side effects
Wrapped (potential) value

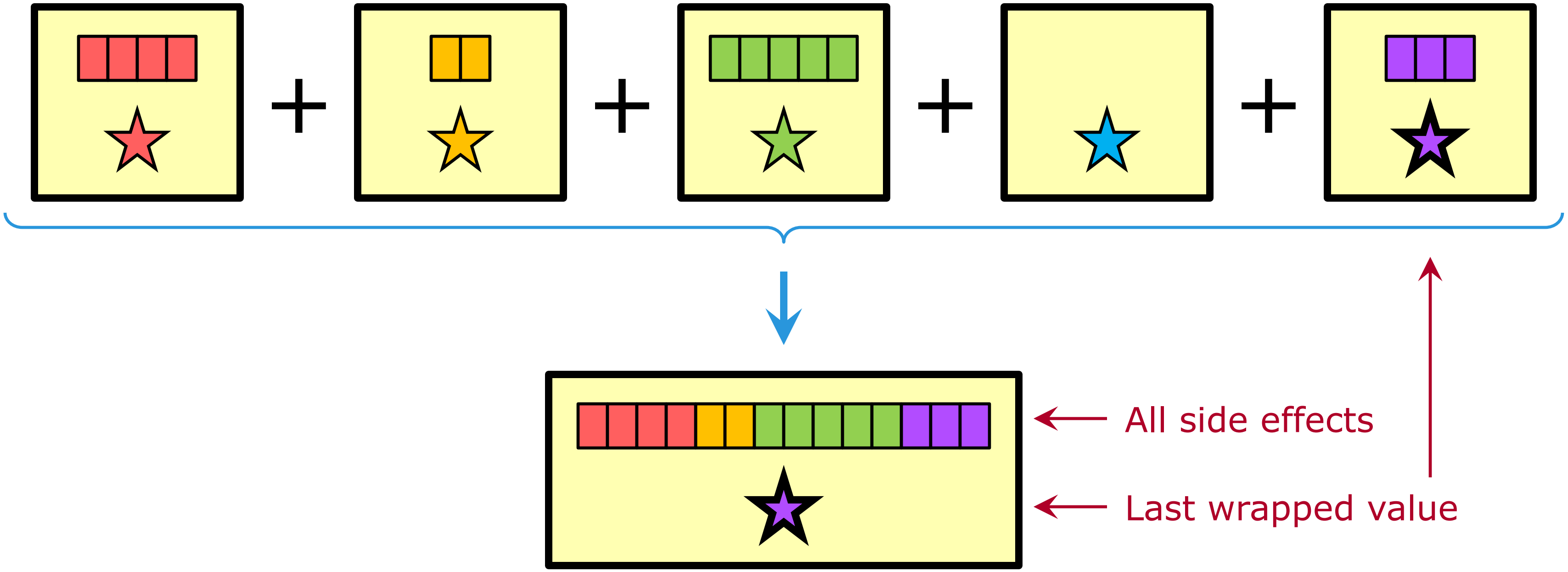
The I/O action returned by `putStr`, illustrated as above.


I/O Action returned by putStr
Print String
Description of side effect: print a String
( )
Wrapped "nothing" value

Review
Haskell: I/O — Simple Output, About I/O Actions [3/4]

Multiple I/O actions can be combined into a single I/O action. In all cases, the resulting I/O action has:

- A description of *all* side effects from the combined I/O actions.
- The wrapped value from the *last* of the combined I/O actions.



Review
Haskell: I/O — Simple Output, About I/O Actions [4/4]

The >> operator combines two I/O actions into one I/O action, as on the previous slide.

```
> putStr "Hello" >> putStrLn " there!"
Hello there!
```

Chain them to combine three or more I/O actions into one.

```
x = putStr "I have " >> putStr (show (73*94*82))
                     >> putStrLn " hamsters."
                     >> putStrLn "Really."
```

```
> x
I have 562684 hamsters.
Really.
```

Haskell: I/O
Simple Input [1/4]

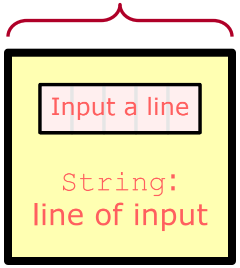
When we do input, we use an I/O action that wraps the value we are inputting.

```
> :t getLine
getLine :: IO String
```

The returned I/O action wraps a String.

getLine returns an I/O action whose side effect is inputting a line of text from the standard input. The wrapped value is a String: the line of text, without the ending newline.

I/O Action returned by getLine

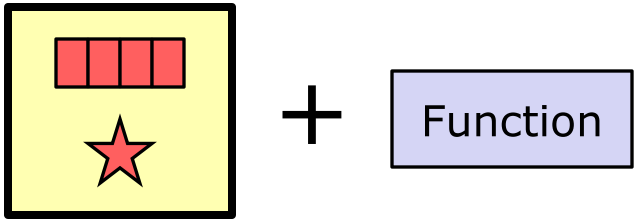


We can access the wrapped String, not by pulling it out of the I/O action, but by pushing a function into the I/O action, using the >>= operator. *See the next slide.*

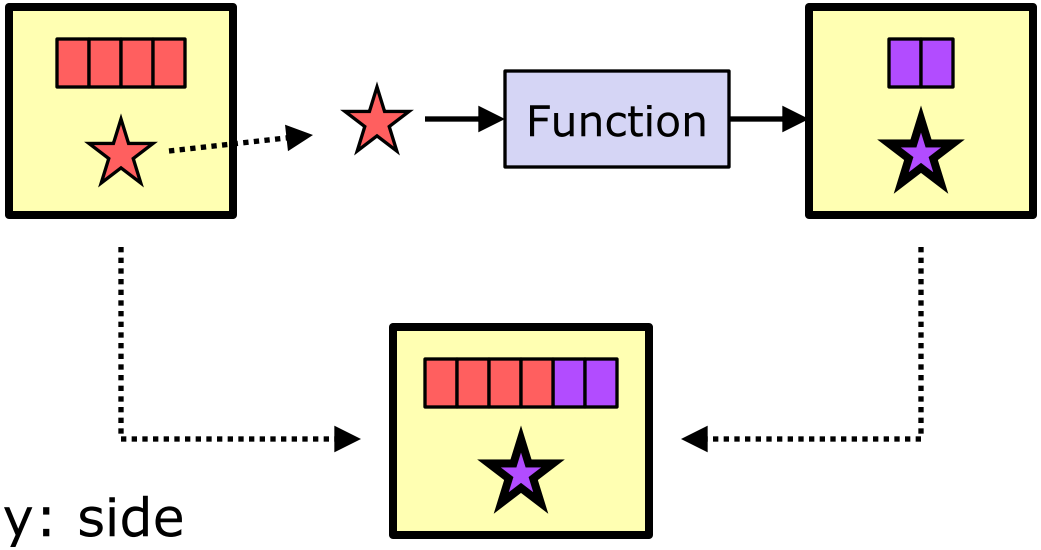
Haskell: I/O
Simple Input [2/4]

Using the >>= operator, we can combine

- an I/O action wrapping a value, and
- a function that takes such a value and returns an I/O action.



The wrapped value is passed to the function, which returns an I/O action.



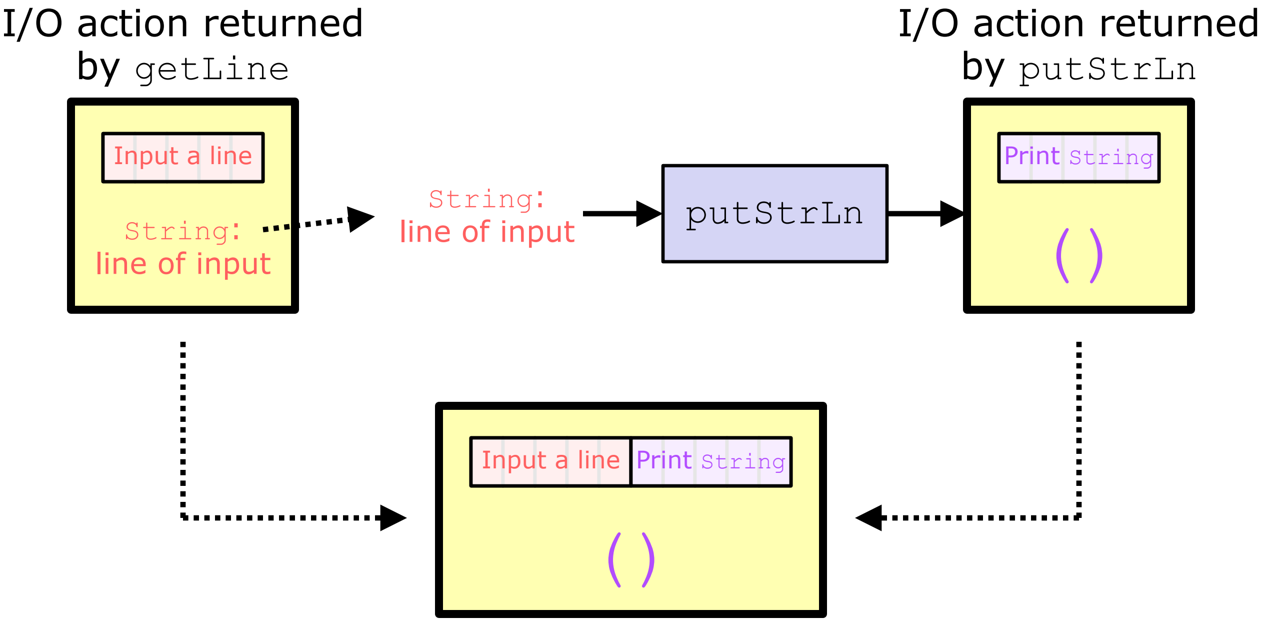
The two I/O actions are combined in the usual way: side effects of both, wrapped value of last.

Haskell: I/O
Simple Input [3/4]

For example, `getLine` returns an I/O action wrapping a `String`. Function `putStrLn` takes a `String` and returns an I/O action.

```
> getLine >>= putStrLn
Howdy!
Howdy!
```

Typed by user



Haskell: I/O
Simple Input [4/4]

I/O actions involving multiple side effects work, too.

```
> putStr "Type something: " >> getLine >>= putStrLn
Type something: I like hamsters!
I like hamsters!
```

We can give the parameter of `putStrLn` a name:

Same as `putStrLn` (right?)

```
> getLine >>= (\ line -> putStrLn line)
Hamsters rule ...
Hamsters rule ...
> getLine >>= (\ line -> putStrLn (reverse line))
... this planet and others like it.
.ti ekil srehto dna tenalp siht ...
```

Haskell: I/O

continued

Haskell: I/O
Do-Expression [1/2]

Haskell's **do-expression** offers a cleaner way to write I/O.

The keyword `do` is followed by an indented block. I/O actions in the block are combined into a single I/O action. Internally, this is done using the `>>` and `>>=` operators.

Using operators:

```
putStr s >> putStrLn t
```

Using a do-expression:

```
do
    putStr s
    putStrLn t
```

Using operators:

```
getLine >>=
    (\ line -> putStrLn line)
```

Using a do-expression:

```
do
    line <- getLine
    putStrLn line
```

This binds the name `line` to the I/O-wrapped value. Variable `line` can then be used in the rest of the do-expression.

Haskell: I/O
Do-Expression [2/2]

TO DO

- Write a function that inputs a line of text and then prints a message indicating the number of characters entered. Use a do-expression.

Done. See `io.hs`*.*

Useful

- Function `hFlush` is given a *handle** to an open file; it returns an I/O action that *flushes*** the file. Do "`hFlush stdout`" after printing a prompt and before doing input, to ensure that the prompt appears before input is entered.
- `hFlush` is a Haskell standard-library function, but it is not in the prelude. To use it in a source file, do "`import System.IO`" near the beginning of the file.

***Handle**: object that identifies and allows access to an open file.

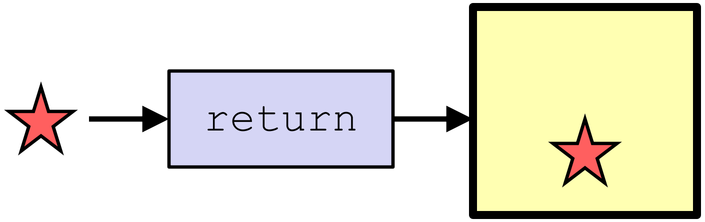
****Flush**: write any buffered characters.

Haskell: I/O
return [1/2]

So far, every I/O action we have used has described one or more side effects. But an I/O action can also involve zero side effects.

To create a no-side-effect I/O action wrapping a particular value, pass the value to `return`.

- `return x` produces a do-nothing I/O action that wraps `x`.
- `return ()` produces a do-nothing, wrap-"nothing" I/O action.



Haskell's `return` does not return! It simply creates a do-nothing I/O action. However, generally we only use `return` as the last thing in a do-expression, so it *feels* like it returns.

Otherwise, it is pointless (right?).

Haskell: I/O
return [2/2]

`getChar` does single-character input. It returns an I/O action wrapping a `Char`.

TO DO

- Using `getChar`, write `myGetLine`, which should do the same thing as `getLine`.
- Write code to use `myGetLine`.

Done. See `io.hs`*.*

Now we have examples of flow of control inside a do-expression: both selection (using `if … then … else`) and repetition (using recursion).

Haskell: I/O
“let” in a Do-Expression [1/2]

One last bit of do-expression syntax remains. We can use `let` *NAME* `=` *EXPR* inside a do-expression to bind a name to a normal value (not I/O-wrapped) for the remainder of the block.

```
foo = do
    let n = 42
    putStr "n = "
    putstrLn $ show n
    let nsq = n*n
    putStr "n + n*n = "
    putStrLn $ show (n+nsq)
```

Haskell: I/O
“let” in a Do-Expression [2/2]

TO DO

- Write a program of the kind that might be assigned early in Computer Science I. Input a number. If the number is zero, then quit. Otherwise, print a message giving some value computed from the number (its square?), and repeat.

Done. See `io.hs`*.*

Remember: a Haskell function must have a consistent return type. If one branch of a selection control structure (pattern matching, `if` … `then` … `else`, guards) returns an I/O action, then the other branches must also return I/O actions.

Haskell: I/O
Notes [1/2]

Summary of Haskell I/O

- Haskell string conversions are largely separate from I/O.
  - `show` converts a value to a `String`.
  - `read` gets a value from a `String`. A type annotation may be required.
- An **I/O action** holds a description of a sequence of side effects plus a wrapped value.
- I/O is performed by returning an I/O action from a program.
- A **do-expression** combines multiple I/O actions into a single I/O action. This construction is syntactic sugar around the `>>` and `>>=` operators.
- Inside a do-expression, *NAME* `<-` *EXPR* binds an identifier to an I/O-wrapped value.
- Inside a do-expression, `let` *NAME* `=` *EXPR* binds an identifier to a non-I/O value.
- `return` *EXPR* creates a do-nothing (no side effects) I/O action wrapping the given value.

Haskell: I/O
Notes [2/2]

If a Haskell program uses values obtained via input, then its behavior is dependent on a side effect. So purity would seem to be compromised. (Right?)

Solution: the `>>=` operator (always used when doing input) actually includes the function passed to it in the returned I/O action—not the result of calling the function, but *the function itself*.

```
getLine >>= putStrLn
```

The resulting I/O action will include function `putStrLn`.

And what is actually included in the returned I/O action is a function to call to run *the entire remainder of the program*.

So when we do input, we might as well be saying, “This program is finished. Now do some input, and pass the value to a *separate program*; here is its code.” And purity is not compromised.

Haskell: Data

Haskell: Data
data Declaration [1/3]

We finish our coverage of Haskell by looking at Haskell’s facilities for defining new types and implementing data structures.

Consider a structure holding information about a product sold in a store. We need to keep the name of the product and the name of the manufacturer.

In C++, we might do something like this:

```
class Product {
private:
    string productName;
    string manufacturerName;
…
};
```

For code from this topic, see `data.hs`*.*

Haskell: Data
data Declaration [2/3]

Here is the more-or-less equivalent Haskell.

```
data Product = Pr String String
  -- product name, manufacturer name
```

`Product` is the name of a new type.

`Pr` is a **constructor** for that type. Values of type `Product` are marked by the fact that they begin with "`Pr`".

For example, here is a (mostly useless) function that takes a `Product` and returns the same `Product`.

```
doNothing :: Product -> Product
doNothing (Pr pn mn) = Pr pn mn
```

Haskell: Data
data Declaration [3/3]

Pattern matching works with constructors.

We can use this to retrieve names from a `Product` object.

```
-- pName - Get product name from a Product
pName :: Product -> String
pName (Pr pn _) = pn

-- mName - Get manufacturer name from a Product
mName :: Product -> String
mName (Pr _ mn) = mn
```

Haskell: Data
Overloading & Typeclasses [1/3]

Suppose we wish to test whether two `Product` values are the same. We consider this to be true if they have the same product name and the same manufacturer name.

```
sameProduct :: Product -> Product -> Bool
sameProduct (Pr pn1 mn1) (Pr pn2 mn2) =
    (pn1 == pn2) && (mn1 == mn2)
```

But it would be nicer if we could use the "`==`" operator.

And in fact we *can* do this.

Overloading in Haskell is done using typeclasses. To overload the "`==`" operator, we use typeclass `Eq`.

Haskell: Data
Overloading & Typeclasses [2/3]

To overload the "`==`" operator for type `Product`, we place this type into the `Eq` typeclass. We want type `Product` to be an **instance** of class `Eq`.

In the **instance declaration**, we provide a definition of the "`==`" operator for `Product`.

```
instance Eq Product  where
    Pr pn1 mn1 == Pr pn2 mn2 =
        (pn1 == pn2) && (mn1 == mn2)
```

Now we can use the "`==`" operator with `Product`.

And we can also use "`/=`" (the inequality operator). Haskell typeclasses typically include default definitions of overloaded functions in terms of others.

Haskell: Data
Overloading & Typeclasses [3/3]

We can similarly provide conversion to `String` for `Product` (overloading function `show`) by placing `Product` into the `Show` typeclass.

```
instance Show Product  where
    show (Pr pn mn) = pn ++ " [made by " ++ mn ++ "]"
```

In GHCi:

```
> Pr "Tide" "Procter & Gamble"
Tide [made by Procter & Gamble]
```

Haskell: Data
TO BE CONTINUED …

Haskell: Data will be continued next time.