Haskell: I/O

CS 331 Programming Languages
Lecture Slides
Friday, February 28, 2025

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

Unit Overview
The Haskell Programming Language

Topics

- ✓ PL feature: type system
- ✓ PL category: functional PLs
- ✓ Introduction to Haskell
- ✓ Haskell: functions
- ✓ Haskell: lists
- ✓ Haskell: flow of control
- Haskell: I/O
- Haskell: data

Review

Haskell: Flow of Control [1/2]

Flow of control refers to the ways a PL determines what code is executed.

For example, flow of control in Lua includes:

- Selection (`if` … `elseif` … `else`).
- Iteration (`for` … `=`, `for` … `in`, `while`, `repeat` … `until`).
- Function calls.
- Coroutines.
- Exceptions.

Haskell has very different flow-of-control facilities from most PLs that are oriented toward imperative programming.

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

Review
Haskell: Flow of Control [2/2]

We will look at a flow-of-control structure called a *do-expression* when we study Haskell I/O. An example:

```
reverseIt = do
    putStr "Type something: "
    hFlush stdout  -- Requires import System.IO
    line <- getLine
    putStr "What you typed, reversed: "
    putStrLn (reverse line)
```

When a program returns this expression's value, this happens:

```
> reverseIt
Type something: Howdy!
What you typed, reversed: !ydwoH
```

Typed by user

Haskell: I/O

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

In many PLs, conversion to & from string is mixed up together with I/O. This makes sense, because when we do text I/O, the values we send and receive must, in the end, be composed of characters.

Some examples are shown below. In each, `n` is an integer variable. Each statement will first convert `n` to a string, then write this string to the standard output.

```
printf("%d", n);        /* C */
cout << n;              // C++
System.out.print(n);    // Java
fmt.Print(n)            // Go
print n                 #  Ruby
io.write(n)             -- Lua
```

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

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

But Haskell keeps string conversion and I/O mostly separate.

```
nstr = show n     -- Convert n to a string
putStr nstr       -- Print the string

putStr $ show n   -- Do both
```

The `$` operator does function application, but it is low precedence (0) and right-associative. So this line is the same as "`putStr (show n)`".

We briefly cover Haskell's string-conversion facilities.
Then we look at Haskell I/O.

To be clear: there is no rule in Haskell that string conversion and I/O must be separated.

You can write a function that does both. However, the two are mostly handled via different types and constructions.

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

A Haskell **typeclass** (or simply **class**) is a collection of types that all implement some particular interface.

Some standard typeclasses:

- `Eq`: **Equality-comparable** types. Every type in class `Eq` has the `==` and `/=` (inequality) operators defined.
- `Ord`: **Orderable** types. Every type in class `Ord` has the various ordered comparison operators defined: `<`, `<=`, `>`, `>=`.
- `Num`: **Numeric** types. Every type in class `Num` has the binary `+`, `-` and `*` operators, along with other things like `abs` (absolute value).

Haskell does overloading *only* via typeclasses.

For example, the types in class `Eq` are the only types for which `==` is defined.

This is what is behind our claim that it is difficult to place Haskell's type checking on the nominal/structural axis.

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

We have seen typeclasses before in contexts like the following Haskell type annotation.

```
blug :: (Eq a, Num a) => a -> a -> Bool
```

The above says that `blug` is a function that takes 2 parameters of type `a` and returns Bool, where `a` can be any numeric type (class `Num`) that is equality-comparable (class `Eq`).

Two standard typeclasses related to string conversion:

- `Show`: **Showable** types. Every type in class `Show` has conversion to `String` using the overloaded function `show`.
- `Read`: **Readable** types. Every type in class `Read` has conversion from `String` using the overloaded function `read`.

Haskell: I/O
String Conversion — show

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

```
> show 3
"3"
> show [True, False]
"[True,False]"
```

Not all types are showable.

```
> square x = x*x
> show square
```

[*Error*]

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

To convert a `String` to a readable type, pass the `String` to `read`.

```
> fivestr = "5"
> 2 + read fivestr
7
```

Type annotations are sometimes needed.

Do you see why no type annotation is needed here?

```
> read fivestr  -- Convert String to ... what?
```

[*Error*]

```
> (read fivestr)::Integer
5
> (read fivestr)::Double
5.0
```

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

```
> (read fivestr)::Integer
5
> (read fivestr)::Double
5.0
```

The above illustrates a noteworthy feature of Haskell.

Both Haskell and C++ support function **overloading**: creating distinct functions with the same name in the same namespace.

In C++ we can overload on the number and types of parameters; we must be able to choose which function to use based only on the number and types of the parameters.

But in Haskell, we can also *overload on the return type*. There are various versions of function `read`; all have the same name, and all take a single `String` parameter. But they have different return types; Haskell can determine which to use based on this.

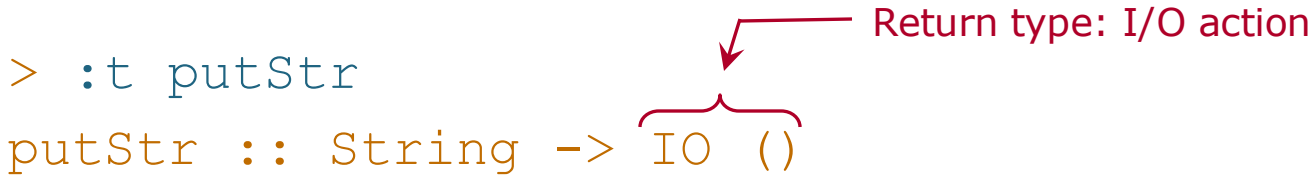
Haskell: I/O
Simple Output [1/4]

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

We do I/O in Haskell as follows: a program's return value includes a description of the side effects the program would like to do. The runtime environment performs the side effects.

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

For example, here is function `putStr`.



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

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

When an expression whose value is an I/O action is entered at the GHCi prompt, the I/O is performed.

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

```
> putStrLn "Hello!"
Hello!
> putStrLn $ show $ map (\ x -> x*x) [1, 2, 3]
[1,4,9]
```

In a complete program, `main` needs to return an I/O action. Here is a Haskell hello-world program.

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

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

The >> operator combines two I/O actions into one I/O action, which describes the side effects of both.

```
> 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 "(Not really.)"
```

```
> x
I have 562684 hamsters.
(Not really.)
```

We will eventually discuss a nicer way to combine I/O actions. But when we use it, this is what is going on under the hood.

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

TO DO

- Write code that does numerical computations and outputs the results. Values stored in variables and/or passed to functions need to be numbers.
- Now do the same thing, but let the values stored in variables and/or passed to functions be I/O actions.

Done. See `io.hs`.

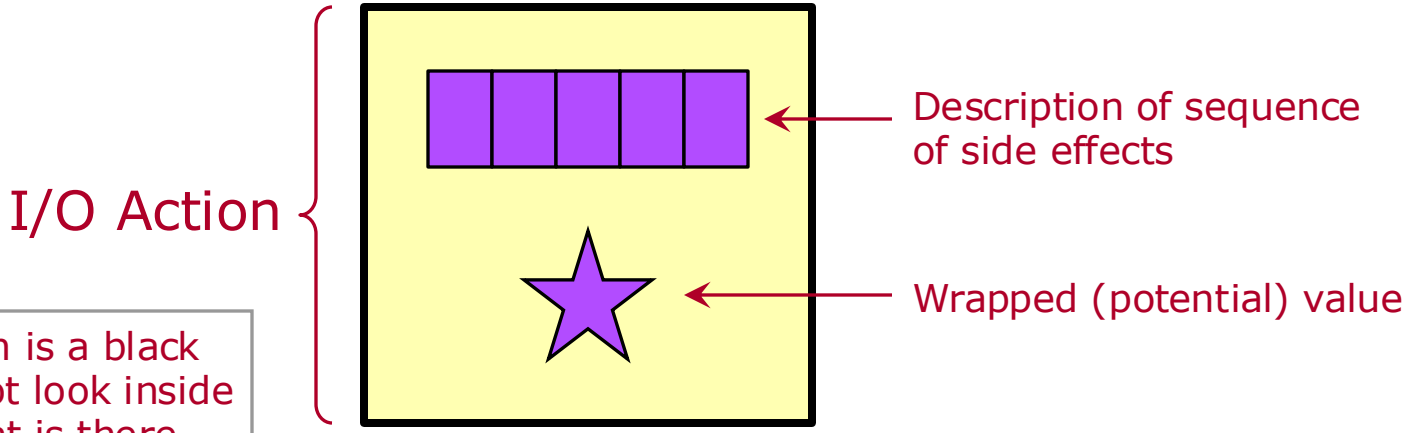
Haskell: I/O
I/O Actions [1/3]

Here is a more complete explanation of an I/O action. It includes:

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

Actually a wrapped *potential* value. Due to laziness, the wrapped value is not evaluated until the I/O action is returned from the program. But we will not need to worry much about this distinction.

I illustrate the above as follows.



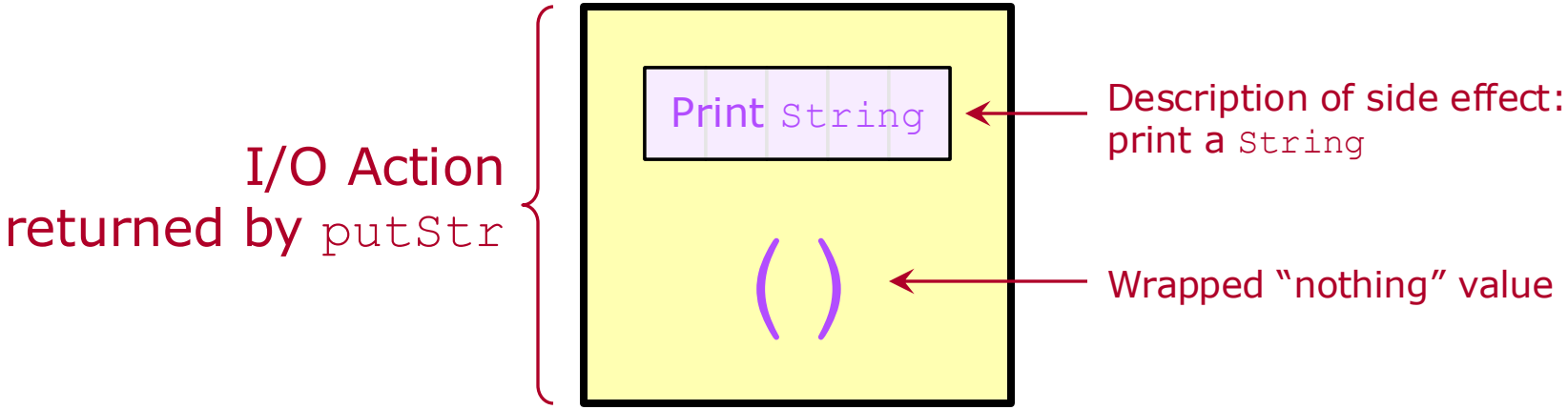
An I/O action is a black box. We cannot look inside and see what is there. (But we can talk about it.)

Haskell: I/O
I/O Actions [2/3]

Recall the "`()`" in the type of `putStr`.

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

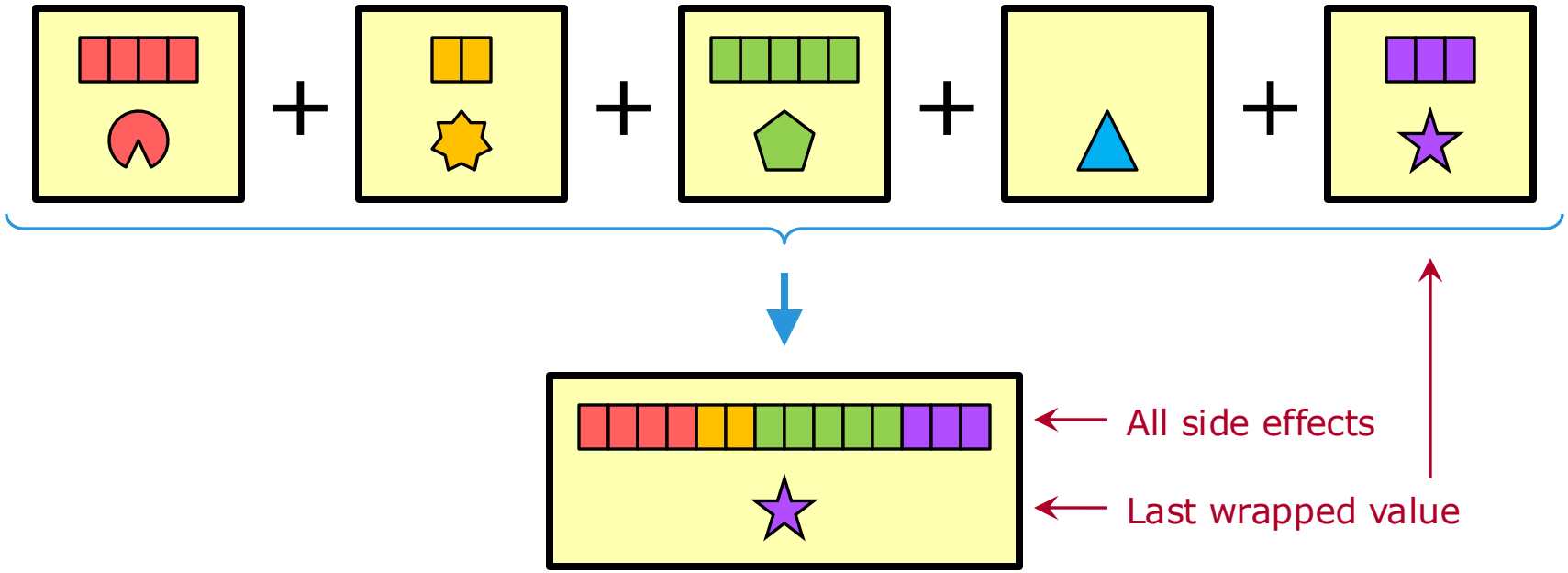
"`()`" means that the I/O action returned by `putStr` wraps a "nothing" value.



I/O Actions [3/3]

There are various ways to combine multiple I/O actions 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.



In particular, the >> operator works this way.

Haskell: I/O
Simple Input [1/5]

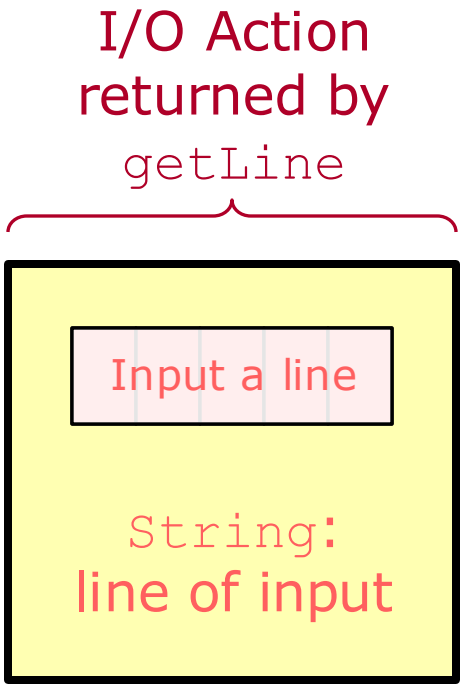
When we do input, we use an I/O action that wraps the value we are inputting.

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

The returned I/O action wraps a `String`.

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

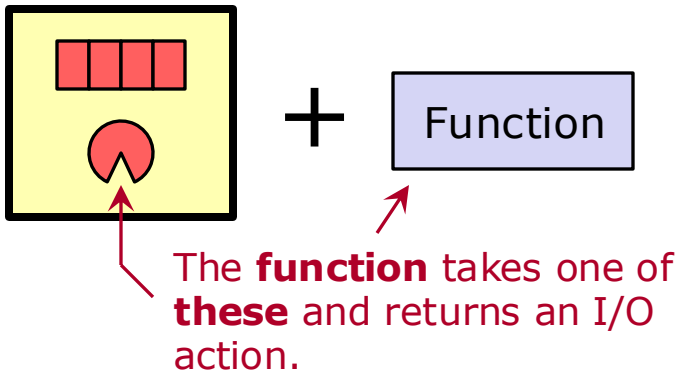
Now, how do we access the wrapped `String`?



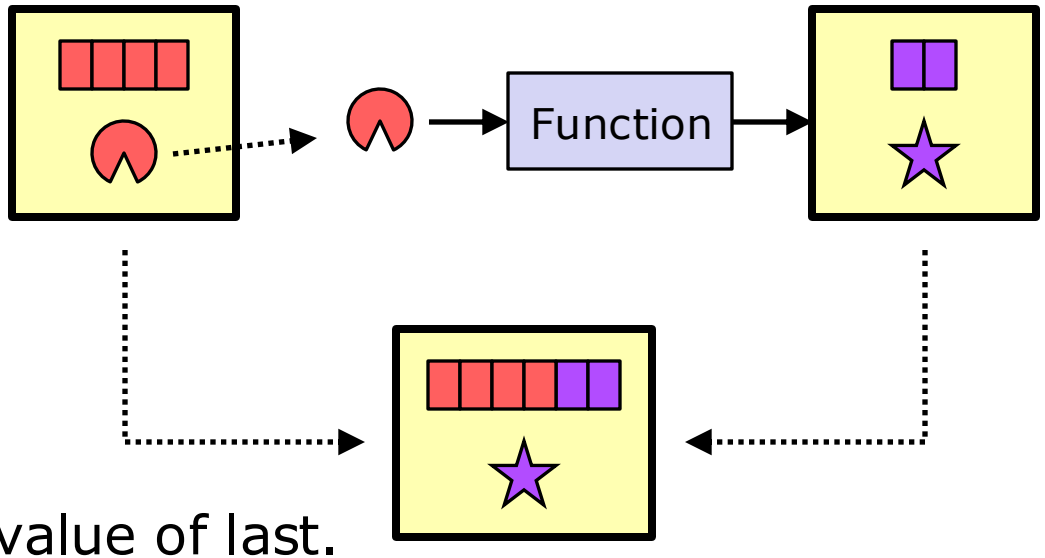
Haskell: I/O
Simple Input [2/5]

The >>= operator has two operands:

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



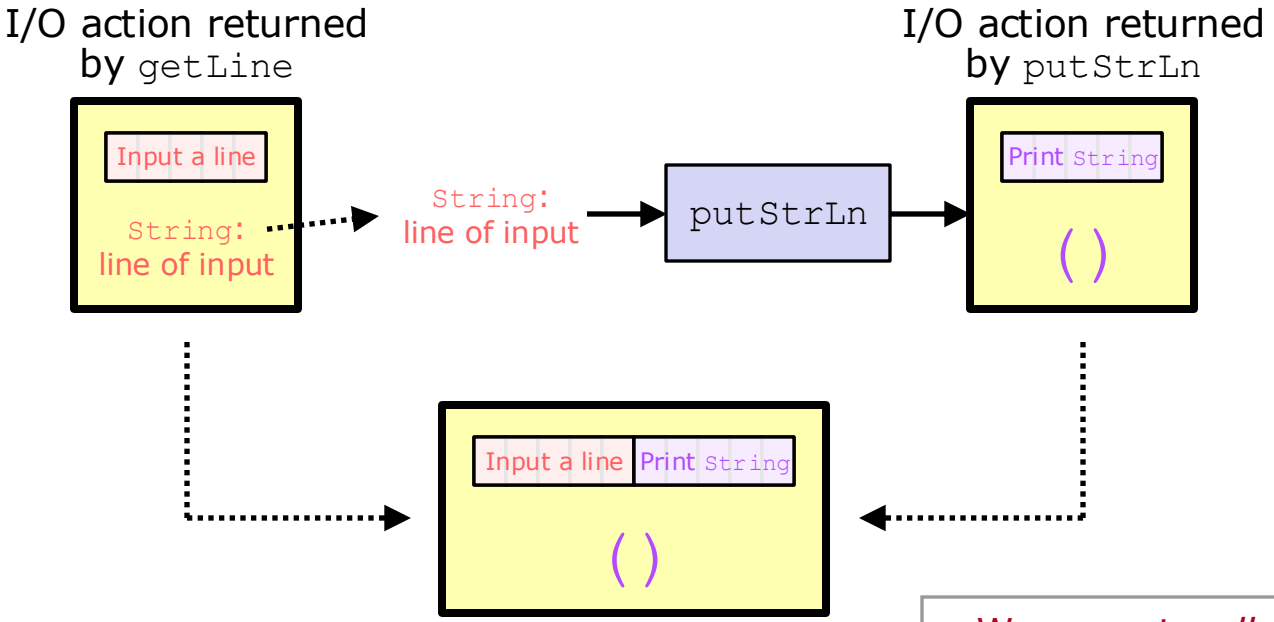
The wrapped value is passed to the function, which returns an I/O action.



The two I/O actions are combined as before: side effects of both, wrapped value of last.

Haskell: I/O
Simple Input [3/5]

For example, `getLine` returns an I/O action wrapping a `String`.
Function `putStrLn` takes a `String` and returns an I/O action.
Put these two together with the `>>=` operator.




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

Typed by user

We cannot *pull* a wrapped value out of an I/O action. Instead, we *push* functions into an I/O action, and pass the wrapped values to them.

Haskell: I/O
Simple Input [4/5]

The >> and >>= operators can be used together:

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

Parentheses are actually unnecessary. The >> and >>= operators have equal precedence and are both left-associative.

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
Simple Input [5/5]

TO DO

- Write some code that does input, and then does output based on this input.

Done. See `io.hs`.

Next: the nicer way to combine I/O actions.

Haskell: I/O
TO BE CONTINUED …

Haskell: I/O will be continued next time.