Mathematically Structured Computer Programs

Sreekar M. Shastry

28 February 2011

```
int f(int n)
{
    return n + CurrentSecond()
}
```

- The type signature, `int f(int n)`, suggests that `f` is a function from the integers to itself, in the mathematical sense.
- But of course it is not.

```
int f(int n)
{
    return n + CurrentSecond()
}
```

- The type signature, `int f(int n)`, suggests that `f` is a function from the integers to itself, in the mathematical sense.
- But of course it is not.

```
int f(int n)
{
    return n + CurrentSecond()
}
```

- The type signature, `int f(int n)`, suggests that `f` is a function from the integers to itself, in the mathematical sense.
- But of course it is not.

Referential Transparency

- In a computer program, a function f is called *referentially transparent* if it always produces the same output Y from the same input X, i.e. if

$$Y = f(X)$$

  is a function in the sense of mathematics.
- This is a fundamental design principle of functional programming languages: Lisp, ML, and most importantly for us
- ***Haskell***
- The paradigm is also called “declarative programming.”

Referential Transparency

- In a computer program, a function f is called *referentially transparent* if it always produces the same output Y from the same input X, i.e. if

$$Y = f(X)$$

  is a function in the sense of mathematics.
- This is a fundamental design principle of functional programming languages: Lisp, ML, and most importantly for us
- ***Haskell***
- The paradigm is also called “declarative programming.”

Referential Transparency

- In a computer program, a function f is called *referentially transparent* if it always produces the same output Y from the same input X, i.e. if

$$Y = f(X)$$

is a function in the sense of mathematics.
- This is a fundamental design principle of functional programming languages: Lisp, ML, and most importantly for us
- ***Haskell***
- The paradigm is also called “declarative programming.”

Referential Transparency

- In a computer program, a function f is called *referentially transparent* if it always produces the same output Y from the same input X, i.e. if

$$Y = f(X)$$

  is a function in the sense of mathematics.
- This is a fundamental design principle of functional programming languages: Lisp, ML, and most importantly for us
- ***Haskell***
- The paradigm is also called “declarative programming.”

Referential Opacity

- Referential opacity, where a function's outputs are not uniquely determined by its inputs is a standard "feature" of procedural programming langauges.
- Also known as "imperative programming."
- Examples: C, C++, Java, Python, C#, Perl, etc.
- The procedure `f(n) = n + CurrentSecond()` may easily be implemented in any of these languages.

Referential Opacity

- Referential opacity, where a function’s outputs are not uniquely determined by its inputs is a standard “feature” of procedural programming langauges.
- Also known as “imperative programming.”
- Examples: C, C++, Java, Python, C#, Perl, etc.
- The procedure `f(n) = n + CurrentSecond()` may easily be implemented in any of these languages.

Referential Opacity

- Referential opacity, where a function's outputs are not uniquely determined by its inputs is a standard "feature" of procedural programming langauges.
- Also known as "imperative programming."
- Examples: C, C++, Java, Python, C#, Perl, etc.
- The procedure `f(n) = n + CurrentSecond()` may easily be implemented in any of these languages.

Referential Opacity

- Referential opacity, where a function’s outputs are not uniquely determined by its inputs is a standard “feature” of procedural programming langauges.
- Also known as “imperative programming.”
- Examples: C, C++, Java, Python, C#, Perl, etc.
- The procedure `f(n) = n + CurrentSecond()` may easily be implemented in any of these languages.

Who cares?

- Referential transparency enables us to reason about program behavior, and is especially helpful for proving program correctness.
- *Imagine trying to prove a theorem of mathematics which involves functions which are not well defined!*

Who cares?

- Referential transparency enables us to reason about program behavior, and is especially helpful for proving program correctness.
- *Imagine trying to prove a theorem of mathematics which involves functions which are not well defined!*

The Factorial Function

Consider the factorial function which we define in an imperative but nevertheless referentially transparent manner

```
int factorial( int n ) {
    if n < 0 then return error
    if n = 0 then return 1

    int x = 1
    for i = 1,2,...,n
        x = x*i

    return x
}
```

The Factorial Function

The standard mathematical definition gives us the referentially transparent definition in a declarative programming language

$$\text{factorial}(n) := \begin{cases} 1 & \text{if } n = 0 \\ n * \text{factorial}(n-1) & \text{if } n > 0 \end{cases}$$

The Factorial Function

The standard mathematical definition gives us the referentially transparent definition in a declarative programming language

$$\text{factorial}(n) := \begin{cases} 1 & \text{if } n = 0 \\ n * \text{factorial}(n-1) & \text{if } n > 0 \end{cases}$$

From mathematics to algorithms and back again...

- If we start with a function from mathematics then we can implement it in a referentially transparent manner in a programming language.
- How about going in the other direction?
- Can we take a standard algorithm from computer science and turn it into a mathematical function?
- Even further: can we model the interaction of the algorithm with the real world in pure mathematics?

From mathematics to algorithms and back again. . .

- If we start with a function from mathematics then we can implement it in a referentially transparent manner in a programming language.
- How about going in the other direction?
- Can we take a standard algorithm from computer science and turn it into a mathematical function?
- Even further: can we model the interaction of the algorithm with the real world in pure mathematics?

From mathematics to algorithms and back again...

- If we start with a function from mathematics then we can implement it in a referentially transparent manner in a programming language.
- How about going in the other direction?
- Can we take a standard algorithm from computer science and turn it into a mathematical function?
- Even further: can we model the interaction of the algorithm with the real world in pure mathematics?

From mathematics to algorithms and back again...

- If we start with a function from mathematics then we can implement it in a referentially transparent manner in a programming language.
- How about going in the other direction?
- Can we take a standard algorithm from computer science and turn it into a mathematical function?
- Even further: can we model the interaction of the algorithm with the real world in pure mathematics?

- For example, can we model our earlier function `f(n) = n + CurrentSecond()` in a referentially transparent manner?

- For example, can we model our earlier function `f(n) = n + CurrentSecond()` in a referentially transparent manner?
- Yes, of course.

- For example, can we model our earlier function `f(n) = n + CurrentSecond()` in a referentially transparent manner?
- Yes, of course.
- We must simply pass in the state of the world as variable...

- For example, can we model our earlier function `f(n) = n + CurrentSecond()` in a referentially transparent manner?
- Yes, of course.
- We must simply pass in the state of the world as variable...
- *so that the reference to the state variable is transparent.*

- For example, can we model our earlier function `f(n) = n + CurrentSecond()` in a referentially transparent manner?
- Yes, of course.
- We must simply pass in the state of the world as variable...
- *so that the reference to the state variable is transparent.*

```
int f(int n, StateOfTheWorld)
{
    return n + CurrentSecond(StateOfTheWorld)
}
```

- For example, can we model our earlier function `f(n) = n + CurrentSecond()` in a referentially transparent manner?
- Yes, of course.
- We must simply pass in the state of the world as variable...
- *so that the reference to the state variable is transparent.*

```
int f(int n, StateOfTheWorld)
{
    return n + CurrentSecond(StateOfTheWorld)
}
```

- ...and likewise for a function which interacts with the user.

- This is what we will do.
- We will see a highly elegant and concise solution to our problem which saves us from *explicitly* passing around cumbersome and potentially very fragile “state of the world” variables.
- The solution to our problem will take us somewhat deeply into the branch of mathematics known as *category theory*.

- This is what we will do.
- We will see a highly elegant and concise solution to our problem which saves us from *explicitly* passing around cumbersome and potentially very fragile “state of the world” variables.
- The solution to our problem will take us somewhat deeply into the branch of mathematics known as *category theory*.

- This is what we will do.
- We will see a highly elegant and concise solution to our problem which saves us from *explicitly* passing around cumbersome and potentially very fragile “state of the world” variables.
- The solution to our problem will take us somewhat deeply into the branch of mathematics known as *category theory*.

A simple program...

```
Main ( Arguments )
    print Quicksort( Arguments )

function Quicksort( Array )
    if length( Array ) <= 1 then return Array
    Pivot := Array[1]
    for each x in Array
        if x <= Pivot then
            append x to LessThanPivotArray
        else
            append x to GreaterThanPivotArray

    return Concatenate( Quicksort( LessThanPivotArray ),
                        Pivot,
                        Quicksort( GreaterThanPivotArray ) )

> ./a.out f g e d c a b
a b c d e f g
```

Visualizing Quicksort

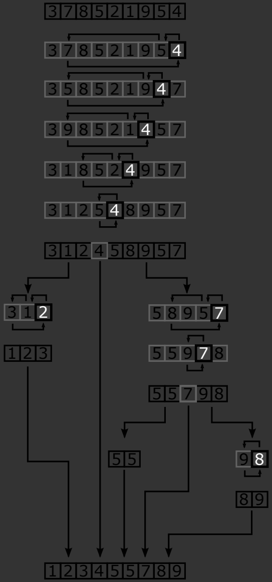

Image courtesy of Wikipedia.

```
liftM             :: (Monad m) => (a -> b) -> m a -> m b
liftM f t         = (\y -> return (f y)) >>> t

q                 :: (Ord a) => [a]->[a]
q []              = []
q (x:xs)          = q [y|y<-xs,y<x] ++ [x] ++ q [y|y<-xs,y>=x]

main = print >>> (liftM q) getArgs
```

```
> runghc quicksort.hs f g e d c a b
["a","b","c","d","e","f","g"]
```

- The preceeding page consisted of three lines of executable Haskell code.
- For the rest of this talk, we will try to understand it, from a mathematical point of view.
- We must review some category theory first...

- The preceeding page consisted of three lines of executable Haskell code.
- For the rest of this talk, we will try to understand it, from a mathematical point of view.
- We must review some category theory first…

- The preceeding page consisted of three lines of executable Haskell code.
- For the rest of this talk, we will try to understand it, from a mathematical point of view.
- We must review some category theory first. . .

Some Category Theory

A *category* $\mathcal{C}$ consists of

- a class $\text{obj}(\mathcal{C})$ of objects
- for all objects $A, B, C, \ldots \in \mathcal{C}$
- a set $\text{Hom}_{\mathcal{C}}(A, B)$ of morphisms
- an identity morphism $\text{id}_A \in \text{Hom}_{\mathcal{C}}(A, A)$
- a composition function

$$\text{Hom}_{\mathcal{C}}(A, B) \times \text{Hom}_{\mathcal{C}}(B, C) \to \text{Hom}_{\mathcal{C}}(A, C)$$

denoted by $g \circ f$ or $f; g$ for

$$A \xrightarrow{f} B \xrightarrow{g} C$$

Some Category Theory

A *category* $\mathcal{C}$ consists of

- ⋆ a class $\mathrm{obj}(\mathcal{C})$ of objects
- ⋆ for all objects $A, B, C, \ldots \in \mathcal{C}$
- ⋆ a set $\mathrm{Hom}_{\mathcal{C}}(A, B)$ of morphisms
- ⋆ an identity morphism $\mathrm{id}_A \in \mathrm{Hom}_{\mathcal{C}}(A, A)$
- ⋆ a composition function

$$\mathrm{Hom}_{\mathcal{C}}(A, B) \times \mathrm{Hom}_{\mathcal{C}}(B, C) \to \mathrm{Hom}_{\mathcal{C}}(A, C)$$

denoted by $g \circ f$ or $f ; g$ for

$$A \xrightarrow{f} B \xrightarrow{g} C$$

Some Category Theory

A *category* $\mathcal{C}$ consists of

- a class $\mathrm{obj}(\mathcal{C})$ of objects
- for all objects $A, B, C, \ldots \in \mathcal{C}$
- a set $\mathrm{Hom}_{\mathcal{C}}(A, B)$ of morphisms
- an identity morphism $\mathrm{id}_A \in \mathrm{Hom}_{\mathcal{C}}(A, A)$
- a composition function

$$\mathrm{Hom}_{\mathcal{C}}(A, B) \times \mathrm{Hom}_{\mathcal{C}}(B, C) \to \mathrm{Hom}_{\mathcal{C}}(A, C)$$

denoted by $g \circ f$ or $f ; g$ for

$$A \xrightarrow{f} B \xrightarrow{g} C$$

Some Category Theory

A *category* $\mathcal{C}$ consists of

- a class $\mathrm{obj}(\mathcal{C})$ of objects
- for all objects $A, B, C, \ldots \in \mathcal{C}$
- a set $\mathrm{Hom}_{\mathcal{C}}(A, B)$ of morphisms
- an identity morphism $\mathrm{id}_A \in \mathrm{Hom}_{\mathcal{C}}(A, A)$
- a composition function

$$\mathrm{Hom}_{\mathcal{C}}(A, B) \times \mathrm{Hom}_{\mathcal{C}}(B, C) \to \mathrm{Hom}_{\mathcal{C}}(A, C)$$

denoted by $g \circ f$ or $f; g$ for

$$A \xrightarrow{f} B \xrightarrow{g} C$$

Some Category Theory

A *category* $\mathcal{C}$ consists of

- a class $\mathrm{obj}(\mathcal{C})$ of objects
- for all objects $A, B, C, \ldots \in \mathcal{C}$
- a set $\mathrm{Hom}_{\mathcal{C}}(A, B)$ of morphisms
- an identity morphism $\mathrm{id}_A \in \mathrm{Hom}_{\mathcal{C}}(A, A)$
- a composition function

$$\mathrm{Hom}_{\mathcal{C}}(A, B) \times \mathrm{Hom}_{\mathcal{C}}(B, C) \to \mathrm{Hom}_{\mathcal{C}}(A, C)$$

denoted by $g \circ f$ or $f; g$ for

$$A \xrightarrow{f} B \xrightarrow{g} C$$

Some Category Theory

A *category* $\mathcal{C}$ consists of

- a class $\mathrm{obj}(\mathcal{C})$ of objects
- for all objects $A, B, C, \ldots \in \mathcal{C}$
- a set $\mathrm{Hom}_{\mathcal{C}}(A, B)$ of morphisms
- an identity morphism $\mathrm{id}_A \in \mathrm{Hom}_{\mathcal{C}}(A, A)$
- a composition function

$$\mathrm{Hom}_{\mathcal{C}}(A, B) \times \mathrm{Hom}_{\mathcal{C}}(B, C) \to \mathrm{Hom}_{\mathcal{C}}(A, C)$$

denoted by $g \circ f$ or $f; g$ for

$$A \xrightarrow{f} B \xrightarrow{g} C$$

Some Category Theory

- such that for

$$A \xrightarrow{f} B \xrightarrow{g} C \xrightarrow{h} D$$

  we have

$$f;(g;h) = (f;g);h$$

- and for $f : A \to B$ we have

$$f; id_B = f = id_A; f$$

Some Category Theory

- such that for

$$A \xrightarrow{f} B \xrightarrow{g} C \xrightarrow{h} D$$

  we have

$$f; (g; h) = (f; g); h$$

- and for $f : A \to B$ we have

$$f; id_B = f = id_A; f$$

Some Category Theory

- The fundamental example is the category of sets, denoted **Sets**
  - objects are sets
  - morphisms are set theoretic functions
  - composition is composition of set functions, etc
- We are interested in the category of Haskell data types $\mathcal{H}$
- We will model $\mathcal{H}$ by **Sets**, i.e. for this talk we define

$$\mathcal{H} := \textbf{Sets}$$

Some Category Theory

- The fundamental example is the category of sets, denoted **Sets**
  - objects are sets
  - morphisms are set theoretic functions
  - composition is composition of set functions, etc
- We are interested in the category of Haskell data types $\mathcal{H}$
- We will model $\mathcal{H}$ by **Sets**, i.e. for this talk we define

$$\mathcal{H} := \textbf{Sets}$$

Some Category Theory

- The fundamental example is the category of sets, denoted **Sets**
  - objects are sets
  - morphisms are set theoretic functions
  - composition is composition of set functions, etc
- We are interested in the category of Haskell data types $\mathcal{H}$
- We will model $\mathcal{H}$ by **Sets**, i.e. for this talk we define

$$\mathcal{H} := \textbf{Sets}$$

The Category of Data Types

- Let us consider $\mathcal{H}$
- An object is to be a data type
- A morphism between data types has as input a data type and produces as output a data type...
- Thus an algorithm written in Haskell is *defined* to be a morphism $f : A \to B$ in $\mathcal{H}$,
- and $A \to B$ is its type signature
- There are further requirements to be an algorithm which we will ignore for this talk...
- Absent a more precise definition, *we know an algorithm when we see one.*

The Category of Data Types

- Let us consider $\mathcal{H}$
- An object is to be a data type
- A morphism between data types has as input a data type and produces as output a data type...
- Thus an algorithm written in Haskell is *defined* to be a morphism $f : A \to B$ in $\mathcal{H}$,
- and $A \to B$ is its type signature
- There are further requirements to be an algorithm which we will ignore for this talk...
- Absent a more precise definition, *we know an algorithm when we see one.*

The Category of Data Types

- Let us consider $\mathcal{H}$
- An object is to be a data type
- A morphism between data types has as input a data type and produces as output a data type...
- Thus an algorithm written in Haskell is *defined* to be a morphism $f : A \to B$ in $\mathcal{H}$,
- and $A \to B$ is its type signature
- There are further requirements to be an algorithm which we will ignore for this talk...
- Absent a more precise definition, *we know an algorithm when we see one.*

The Category of Data Types

- Let us consider $\mathcal{H}$
- An object is to be a data type
- A morphism between data types has as input a data type and produces as output a data type...
- Thus an algorithm written in Haskell is *defined* to be a morphism $f : A \to B$ in $\mathcal{H}$,
- and $A \to B$ is its type signature
- There are further requirements to be an algorithm which we will ignore for this talk...
- Absent a more precise definition, *we know an algorithm when we see one.*

The Category of Data Types

- Let us consider $\mathcal{H}$
- An object is to be a data type
- A morphism between data types has as input a data type and produces as output a data type...
- Thus an algorithm written in Haskell is *defined* to be a morphism $f : A \to B$ in $\mathcal{H}$,
- and $A \to B$ is its type signature
- There are further requirements to be an algorithm which we will ignore for this talk...
- Absent a more precise definition, *we know an algorithm when we see one.*

The Category of Data Types

- Let us consider $\mathcal{H}$
- An object is to be a data type
- A morphism between data types has as input a data type and produces as output a data type...
- Thus an algorithm written in Haskell is *defined* to be a morphism $f : A \rightarrow B$ in $\mathcal{H}$,
- and $A \rightarrow B$ is its type signature
- There are further requirements to be an algorithm which we will ignore for this talk...
- Absent a more precise definition, *we know an algorithm when we see one.*

The Category of Data Types

- Let us consider $\mathcal{H}$
- An object is to be a data type
- A morphism between data types has as input a data type and produces as output a data type...
- Thus an algorithm written in Haskell is *defined* to be a morphism $f : A \to B$ in $\mathcal{H}$,
- and $A \to B$ is its type signature
- There are further requirements to be an algorithm which we will ignore for this talk...
- Absent a more precise definition, *we know an algorithm when we see one.*

The Category of Data Types

- Examples of data types:
- $\mathbb{Z}$, the integers
- $L(\mathbb{Z})$:= the set of lists of integers
- $L(A)$:= the set of lists of type A for $A \in \mathcal{H}$

$$L(A) := \bigcup_{n \geqslant 0} A^n$$
$$= \{(a_1, \ldots, a_m) : m < \infty, a_i \in A, \forall i = 1, \ldots m\}.$$

- Strings := Lists of characters
- Binary trees with leaves of type $A \in \mathcal{H}$

The Category of Data Types

- Examples of data types:
- $\mathbb{Z}$, the integers
- $L(\mathbb{Z})$:= the set of lists of integers
- $L(A)$:= the set of lists of type A for $A \in \mathcal{H}$

$$L(A) := \bigcup_{n \geqslant 0} A^n$$
$$= \{(a_1, \dots, a_m) : m < \infty, a_i \in A, \forall i = 1, \dots m\}.$$

- Strings := Lists of characters
- Binary trees with leaves of type $A \in \mathcal{H}$

The Category of Data Types

- Examples of data types:
- $\mathbb{Z}$, the integers
- $L(\mathbb{Z})$:= the set of lists of integers
- $L(A)$:= the set of lists of type A for $A \in \mathcal{H}$

$$
\begin{aligned}
L(A) &:= \bigcup_{n \geqslant 0} A^n \\
&= \{(a_1, \ldots, a_m) : m < \infty, a_i \in A, \forall i = 1, \ldots m\}.
\end{aligned}
$$

- Strings := Lists of characters
- Binary trees with leaves of type $A \in \mathcal{H}$

The Category of Data Types

- Examples of data types:
- $\mathbb{Z}$, the integers
- $L(\mathbb{Z})$:= the set of lists of integers
- $L(A)$:= the set of lists of type A for $A \in \mathcal{H}$

$$L(A) := \bigcup_{n \geqslant 0} A^n$$
$$= \{(a_1, \ldots, a_m) : m < \infty, a_i \in A, \forall i = 1, \ldots m\}.$$

- Strings := Lists of characters
- Binary trees with leaves of type $A \in \mathcal{H}$

The Category of Data Types

- Examples of data types:
- $\mathbb{Z}$, the integers
- $L(\mathbb{Z})$:= the set of lists of integers
- $L(A)$:= the set of lists of type A for $A \in \mathcal{H}$

$$L(A) := \bigcup_{n \geqslant 0} A^n$$
$$= \{(a_1, \dots, a_m) : m < \infty, a_i \in A, \forall i = 1, \dots m\}.$$

- Strings := Lists of characters
- Binary trees with leaves of type $A \in \mathcal{H}$

The Category of Data Types

- Examples of data types:
- $\mathbb{Z}$, the integers
- $L(\mathbb{Z})$:= the set of lists of integers
- $L(A)$:= the set of lists of type A for $A \in \mathcal{H}$

$$
\begin{aligned}
L(A) &:= \bigcup_{n \geqslant 0} A^n \\
&= \{(a_1, \ldots, a_m) : m < \infty, a_i \in A, \forall i = 1, \ldots m\}.
\end{aligned}
$$

- Strings := Lists of characters
- Binary trees with leaves of type $A \in \mathcal{H}$

The Category of Data Types

- We see that L assigns to a data type A a new data type, the set of lists on A,
- thus is some sort of map from $\mathcal{H}$ to $\mathcal{H}$
- We may say that "List" is a parametrized data type

The Category of Data Types

- We see that L assigns to a data type A a new data type, the set of lists on A,
- thus is some sort of map from $\mathcal{H}$ to $\mathcal{H}$
- We may say that "List" is a parametrized data type

The Category of Data Types

- We see that L assigns to a data type A a new data type, the set of lists on A,
- thus is some sort of map from $\mathcal{H}$ to $\mathcal{H}$
- We may say that "List" is a parametrized data type

Some Category Theory

Definition

A functor $F : \mathcal{C} \to \mathcal{C}'$ from a category $\mathcal{C}$ to a category $\mathcal{C}'$ is a rule which
– given $A \in \mathcal{C}$ produces $F(A) \in \mathcal{C}'$ and
– given $f : A \to B$ in $\mathcal{C}$, produces $F(f) : F(A) \to F(B)$ in $\mathcal{C}'$
– and satisfies various axioms, for instance $F(f;g) = F(f);F(g)$

Some Category Theory

Definition

Given functors $F : \mathcal{C} \to \mathcal{C}'$ and $G : \mathcal{C}' \to \mathcal{C}''$ we may compose them to obtain

$$F; G : \mathcal{C} \to \mathcal{C}''.$$

Thus we have the notion of $\mathrm{End}(\mathcal{C})$, which is the category of all endofunctors of $\mathcal{C}$:

– the objects are functors $F : \mathcal{C} \to \mathcal{C}$ (known as endofunctors)

– the morphisms are natural transformations of functors

Some Category Theory

Definition

A monad over $\mathcal{C}$ is a monoid in $\mathrm{End}(\mathcal{C})$. By this we mean a functor $T \in \mathrm{End}(\mathcal{C})$ together with natural transformations $\mu : T \times T \to T$ and $\mu : \mathrm{id}_{\mathcal{C}} \to T$ which satisfy the associativity and unit axioms of a monoid

$$
\begin{tikzcd}
T \times T \times T \arrow[r, "\mu \times \mathrm{id}_T"] \arrow[d, "\mathrm{id}_T \times \mu"'] & T \times T \arrow[d, "\mu"] \\
T \times T \arrow[r, "\mu"'] & T
\end{tikzcd}
\qquad
\begin{tikzcd}
T \times \mathrm{id}_{\mathcal{C}} \arrow[r, "\mathrm{id}_T \times \eta"] \arrow[dr, equal] & T \times T \arrow[d, "\mu"] & \mathrm{id}_{\mathcal{C}} \times T \arrow[l, "\eta \times \mathrm{id}_T"'] \arrow[dl, equal] \\
& T &
\end{tikzcd}
$$

(here "$\times$" indicates the composition of functors, which is the product structure in $\mathrm{End}(\mathcal{C})$.)

Some Category Theory

Definition

A Kleisli triple over $\mathcal{C}$ is a triple

$$(T, \eta, (\cdot)^*)$$

where

(1) $T : |\mathcal{C}| \to |\mathcal{C}|$ is an assignment on objects

(2) $\eta_A : A \to TA$ is a morphism in $\mathcal{C}$

(3) $f^* : TA \to TB$ for given $f : A \to TB$

s.t. the following hold:

(a) $\eta_A^* = \mathrm{id}_{TA}$

(b) $\eta_A ; f^* = f$ for $f : A \to TB$

(c) $f^* ; g^* = (f ; g^*)^*$ for $f : A \to TB$ and $g : B \to TC$

Some Category Theory

Theorem

The notions of Kleisli triple and monad are equivalent.

Proof.

Given a Kleisli triple $(T, \eta, (\cdot)^*)$ the corresponding monad is (T, η, μ) where we make T into an endofunctor by defining for $f : A \to B$

$$Tf := (f; \eta_B)^* \text{ and } \mu_A := \mathrm{id}^*_{TA}.$$

Conversely, given a monad (T, η, μ) we define a Kleisli triple by restricting the functor T to objects and for $f : A \to TB$ we put

$$f^* := (Tf); \mu_B.$$

$\square$

Some Category Theory

Theorem

The notions of Kleisli triple and monad are equivalent.

Proof.

Given a Kleisli triple $(T, \eta, (\cdot)^*)$ the corresponding monad is (T, η, μ) where we make T into an endofunctor by defining for $f : A \to B$

$$Tf := (f; \eta_B)^* \text{ and } \mu_A := \mathrm{id}^*_{TA}.$$

Conversely, given a monad (T, η, μ) we define a Kleisli triple by restricting the functor T to objects and for $f : A \to TB$ we put

$$f^* := (Tf); \mu_B.$$

Some Category Theory

Theorem

The notions of Kleisli triple and monad are equivalent.

Proof.

Given a Kleisli triple $(T, \eta, (\cdot)^*)$ the corresponding monad is (T, η, μ) where we make T into an endofunctor by defining for $f : A \to B$

$$Tf := (f; \eta_B)^* \text{ and } \mu_A := \mathrm{id}^*_{TA}.$$

Conversely, given a monad (T, η, μ) we define a Kleisli triple by restricting the functor T to objects and for $f : A \to TB$ we put

$$f^* := (Tf); \mu_B.$$

$\square$

The List Monad

- Let us see that the endofunctor $L \in \mathrm{End}(\mathcal{H})$ is actually a monad.
- $L(A)$:= the set of lists of type A for $A \in \mathcal{H}$

$$\begin{aligned} L(A) &:= \bigcup_{n \geqslant 0} A^n \\ &= \{(a_1, \ldots, a_m) : m < \infty, a_i \in A, \forall i = 1, \ldots m\}. \end{aligned}$$

The List Monad

- Let us see that the endofunctor $L \in \mathrm{End}(\mathcal{H})$ is actually a monad.
- $L(A)$:= the set of lists of type A for $A \in \mathcal{H}$

$$\begin{aligned} L(A) &:= \bigcup_{n \geqslant 0} A^n \\ &= \{(a_1, \ldots, a_m) : m < \infty, a_i \in A, \forall i = 1, \ldots m\}. \end{aligned}$$

The List Monad

- An example will tell us how to define the monadic "multiplication" and "identity"
  $\mu_A : (L \times L)(A) := L(L(A)) \to L(A)$ and $\eta_A : A \to L(A)$
- Take $A := \mathbb{Z}$. Then

$$((1,2),(3),(4,5)) \xrightarrow{\mu_{\mathbb{Z}}} (1,2,3,4,5)$$

removes a layer of parentheses and

$$n \xrightarrow{\eta_{\mathbb{Z}}} (n)$$

is the list consisting of a single element.

The List Monad

- An example will tell us how to define the monadic "multiplication" and "identity" $\mu_A : (L \times L)(A) := L(L(A)) \to L(A)$ and $\eta_A : A \to L(A)$
- Take $A := \mathbb{Z}$. Then

$$((1, 2), (3), (4, 5)) \xrightarrow{\mu_{\mathbb{Z}}} (1, 2, 3, 4, 5)$$

removes a layer of parentheses and

$$n \xrightarrow{\eta_{\mathbb{Z}}} (n)$$

is the list consisting of a single element.

The List Monad

- An example will tell us how to define the monadic "multiplication" and "identity" $\mu_A : (L \times L)(A) := L(L(A)) \to L(A)$ and $\eta_A : A \to L(A)$
- Take $A := \mathbb{Z}$. Then

$$((1,2),(3),(4,5)) \xrightarrow{\mu_{\mathbb{Z}}} (1,2,3,4,5)$$

removes a layer of parentheses and

$$n \xrightarrow{\eta_{\mathbb{Z}}} (n)$$

is the list consisting of a single element.

The List Monad

- An example will tell us how to define the monadic "multiplication" and "identity" $\mu_A : (L \times L)(A) := L(L(A)) \to L(A)$ and $\eta_A : A \to L(A)$
- Take $A := \mathbb{Z}$. Then

$$((1,2),(3),(4,5)) \xrightarrow{\mu_{\mathbb{Z}}} (1,2,3,4,5)$$

removes a layer of parentheses and

$$n \xrightarrow{\eta_{\mathbb{Z}}} (n)$$

is the list consisting of a single element.

Coding the List Monad

```
instance Monad List where
    return x = [x]
    f >>> xs  = (concat . fmap f) xs
```

- `return x` is $\eta_A(x)$
- `xs` is a list, so how to define `f >>> xs`?
- given $f : A \to L(B)$ the Kleisli star gives us $f^* : L(A) \to L(B)$,
- `concat` = μ_A,
- `fmap f` = $L(f)$,

Coding the List Monad

```
instance Monad List where
    return x = [x]
    f >>> xs  = (concat . fmap f) xs
```

- `return x` is $\eta_A(x)$
- `xs` is a list, so how to define `f >>> xs`?
- given $f : A \to L(B)$ the Kleisli star gives us $f^* : L(A) \to L(B)$,
- `concat` = μ_A,
- `fmap f` = $L(f)$,

Coding the List Monad

```
instance Monad List where
    return x = [x]
    f >>> xs  = (concat . fmap f) xs
```

- `return x` is $\eta_A(x)$
- `xs` is a list, so how to define `f >>> xs`?
- given $f : A \to L(B)$ the Kleisli star gives us $f^* : L(A) \to L(B)$,
- `concat` $= \mu_A$,
- `fmap f` $= L(f)$,

Coding the List Monad

```
instance Monad List where
    return x = [x]
    f >>> xs  = (concat . fmap f) xs
```

- `return x` is $\eta_A(x)$
- `xs` is a list, so how to define `f >>> xs`?
- given $f : A \to L(B)$ the Kleisli star gives us $f^* : L(A) \to L(B)$,
- `concat` = μ_A,
- `fmap f` = $L(f)$,

Coding the List Monad

```
instance Monad List where
    return x = [x]
    f >>> xs  = (concat . fmap f) xs
```

- `return x` is $\eta_A(x)$
- `xs` is a list, so how to define `f >>> xs`?
- given $f : A \to L(B)$ the Kleisli star gives us $f^* : L(A) \to L(B)$,
- `concat` $= \mu_A$,
- `fmap f` $= L(f)$,

Coding the List Monad

```
instance Monad List where
    return x = [x]
    f >>> xs  = (concat . fmap f) xs
```

- `return x` is $\eta_A(x)$
- `xs` is a list, so how to define `f >>> xs`?
- given $f : A \rightarrow L(B)$ the Kleisli star gives us $f^* : L(A) \rightarrow L(B)$,
- `concat` $= \mu_A$,
- `fmap f` $= L(f)$,

Coding the List Monad

- thus

$$(\text{ concat . fmap f }) = \mu_B \circ L(f) = L(f); \mu_B =: f^*$$

- and

$$\begin{aligned} \text{f} >>> \text{xs} &:= f^*(xs) \\ &= \mu_B(L(f)(xs)) \\ &= \mu_B((f(y) : y \in xs)) \\ &= \text{concat}(f(y) : y \in xs) \end{aligned}$$

Coding the List Monad

- thus

$$(\text{ concat . fmap f }) = \mu_B \circ L(f) = L(f);\mu_B =: f^*$$

- and

$$\begin{aligned}
\text{f} >>> \text{xs} &:= f^*(xs) \\
&= \mu_B(L(f)(xs)) \\
&= \mu_B((f(y) : y \in xs)) \\
&= \mathtt{concat}(f(y) : y \in xs)
\end{aligned}$$

Coding the List Monad

- An example:
- $f : \mathbb{Z} \to L(\mathbb{Z})$
- `f n = [n,n^2]`
- `f >>> [1..5] =`
- `(concat . fmap f) [1..5] =`
- `concat (fmap f [1..5]) =`
- `concat [[1,1],[2,4],[3,9],[4,16],[5,25]] =`
- `[1,1,2,4,3,9,4,16,5,25]`

Coding the List Monad

- An example:
- $f : \mathbb{Z} \to L(\mathbb{Z})$
- `f n = [n,n^2]`
- `f >>> [1..5] =`
- `(concat . fmap f) [1..5] =`
- `concat (fmap f [1..5]) =`
- `concat [[1,1],[2,4],[3,9],[4,16],[5,25]] =`
- `[1,1,2,4,3,9,4,16,5,25]`

Coding the List Monad

- An example:
- $f : \mathbb{Z} \to L(\mathbb{Z})$
- `f n = [n,n^2]`
- `f >>> [1..5] =`
- `(concat . fmap f) [1..5] =`
- `concat (fmap f [1..5]) =`
- `concat [[1,1],[2,4],[3,9],[4,16],[5,25]] =`
- `[1,1,2,4,3,9,4,16,5,25]`

Coding the List Monad

- An example:
- $f : \mathbb{Z} \to L(\mathbb{Z})$
- `f n = [n,n^2]`
- `f >>> [1..5] =`
- `(concat . fmap f) [1..5] =`
- `concat (fmap f [1..5]) =`
- `concat [[1,1],[2,4],[3,9],[4,16],[5,25]] =`
- `[1,1,2,4,3,9,4,16,5,25]`

Coding the List Monad

- An example:
- $f : \mathbb{Z} \to L(\mathbb{Z})$
- `f n = [n,n^2]`
- `f >>> [1..5] =`
- `(concat . fmap f) [1..5] =`
- `concat (fmap f [1..5]) =`
- `concat [[1,1],[2,4],[3,9],[4,16],[5,25]] =`
- `[1,1,2,4,3,9,4,16,5,25]`

Coding the List Monad

- An example:
- $f : \mathbb{Z} \to L(\mathbb{Z})$
- `f n = [n,n^2]`
- `f >>> [1..5] =`
- `(concat . fmap f) [1..5] =`
- `concat (fmap f [1..5]) =`
- `concat [[1,1],[2,4],[3,9],[4,16],[5,25]] =`
- `[1,1,2,4,3,9,4,16,5,25]`

Coding the List Monad

- An example:
- $f : \mathbb{Z} \to L(\mathbb{Z})$
- `f n = [n,n^2]`
- `f >>> [1..5] =`
- `(concat . fmap f) [1..5] =`
- `concat (fmap f [1..5]) =`
- `concat [[1,1],[2,4],[3,9],[4,16],[5,25]] =`
- `[1,1,2,4,3,9,4,16,5,25]`

The State Monad

- We model the current state of the world as a pair of strings $(\texttt{is}, \texttt{os}) \in \mathcal{S} \times \mathcal{S}$ where
- $\mathcal{S}$ is the set of all strings
- `is` = characters waiting to be read in the input stream
- `os` = characters already written to the output stream
- We define the state monad as an endofunctor of $\mathcal{H}$, $T : \mathcal{H} \to \mathcal{H}$ to be

$$T(A) := \mathbf{Hom}(\mathcal{S} \times \mathcal{S}, \mathcal{S} \times \mathcal{S} \times A)$$

- One can show that it forms a monad

The State Monad

- We model the current state of the world as a pair of strings $(\mathtt{is}, \mathtt{os}) \in \mathcal{S} \times \mathcal{S}$ where
- $\mathcal{S}$ is the set of all strings
- `is` = characters waiting to be read in the input stream
- `os` = characters already written to the output stream
- We define the state monad as an endofunctor of $\mathcal{H}$, $T : \mathcal{H} \to \mathcal{H}$ to be

$$T(A) := \mathbf{Hom}(\mathcal{S} \times \mathcal{S}, \mathcal{S} \times \mathcal{S} \times A)$$

- One can show that it forms a monad

The State Monad

- We model the current state of the world as a pair of strings $(\texttt{is}, \texttt{os}) \in \mathcal{S} \times \mathcal{S}$ where
- $\mathcal{S}$ is the set of all strings
- `is` = characters waiting to be read in the input stream
- os = characters already written to the output stream
- We define the state monad as an endofunctor of $\mathcal{H}$, $T : \mathcal{H} \to \mathcal{H}$ to be

$$T(A) := \mathbf{Hom}(\mathcal{S} \times \mathcal{S}, \mathcal{S} \times \mathcal{S} \times A)$$

- One can show that it forms a monad

The State Monad

- We model the current state of the world as a pair of strings $(\texttt{is}, \texttt{os}) \in \mathcal{S} \times \mathcal{S}$ where
- $\mathcal{S}$ is the set of all strings
- `is` = characters waiting to be read in the input stream
- `os` = characters already written to the output stream
- We define the state monad as an endofunctor of $\mathcal{H}$, $T : \mathcal{H} \to \mathcal{H}$ to be

$$T(A) := \mathbf{Hom}(\mathcal{S} \times \mathcal{S}, \mathcal{S} \times \mathcal{S} \times A)$$

- One can show that it forms a monad

The State Monad

- We model the current state of the world as a pair of strings $(\texttt{is}, \texttt{os}) \in \mathcal{S} \times \mathcal{S}$ where
- $\mathcal{S}$ is the set of all strings
- `is` = characters waiting to be read in the input stream
- `os` = characters already written to the output stream
- We define the state monad as an endofunctor of $\mathcal{H}$, $T : \mathcal{H} \to \mathcal{H}$ to be

$$T(A) := \mathrm{Hom}(\mathcal{S} \times \mathcal{S}, \mathcal{S} \times \mathcal{S} \times A)$$

- One can show that it forms a monad

The State Monad

- We model the current state of the world as a pair of strings $(\texttt{is}, \texttt{os}) \in \mathcal{S} \times \mathcal{S}$ where
- $\mathcal{S}$ is the set of all strings
- `is` = characters waiting to be read in the input stream
- `os` = characters already written to the output stream
- We define the state monad as an endofunctor of $\mathcal{H}$, $T : \mathcal{H} \to \mathcal{H}$ to be

$$T(A) := \mathrm{Hom}(\mathcal{S} \times \mathcal{S}, \mathcal{S} \times \mathcal{S} \times A)$$

- One can show that it forms a monad

The State Monad

- Thus a $f \in T(A)$ assigns to every possible initial state of the world $(\mathtt{i}, \mathtt{o})$ a final state $(\mathtt{i}', \mathtt{o}', \alpha)$
- We view f as the action of a single step of an algorithm
- In case we are reading input, we think of α as the value read off of the initial input stream, resulting in the final input stream and output stream
- In case of printing output, α would be an empty value
- Example of reading input
  `i = "def", o = "cba"`
  `i' = "ef", o' = "cba",` α `= "d"`
- As described earlier, it is in this way that we pass around the state of the world as a variable, in a referentially transparent manner.

The State Monad

- Thus a $f \in T(A)$ assigns to every possible initial state of the world $(\mathtt{i}, \mathtt{o})$ a final state $(\mathtt{i}', \mathtt{o}', \alpha)$
- We view f as the action of a single step of an algorithm
- In case we are reading input, we think of α as the value read off of the initial input stream, resulting in the final input stream and output stream
- In case of printing output, α would be an empty value
- Example of reading input
  `i = "def", o = "cba"`
  `i' = "ef", o' = "cba",` α `= "d"`
- As described earlier, it is in this way that we pass around the state of the world as a variable, in a referentially transparent manner.

The State Monad

- Thus a $f \in T(A)$ assigns to every possible initial state of the world $(\mathtt{i}, \mathtt{o})$ a final state $(\mathtt{i}', \mathtt{o}', \alpha)$
- We view f as the action of a single step of an algorithm
- In case we are reading input, we think of α as the value read off of the initial input stream, resulting in the final input stream and output stream
- In case of printing output, α would be an empty value
- Example of reading input
  `i = "def", o = "cba"`
  `i' = "ef", o' = "cba",` α `= "d"`
- As described earlier, it is in this way that we pass around the state of the world as a variable, in a referentially transparent manner.

The State Monad

- Thus a $f \in T(A)$ assigns to every possible initial state of the world $(\mathtt{i}, \mathtt{o})$ a final state $(\mathtt{i}', \mathtt{o}', \alpha)$
- We view f as the action of a single step of an algorithm
- In case we are reading input, we think of α as the value read off of the initial input stream, resulting in the final input stream and output stream
- In case of printing output, α would be an empty value
- Example of reading input
  `i = "def", o = "cba"`
  `i' = "ef", o' = "cba",` α `= "d"`
- As described earlier, it is in this way that we pass around the state of the world as a variable, in a referentially transparent manner.

The State Monad

- Thus a $f \in T(A)$ assigns to every possible initial state of the world $(\mathtt{i}, \mathtt{o})$ a final state $(\mathtt{i}', \mathtt{o}', \alpha)$
- We view f as the action of a single step of an algorithm
- In case we are reading input, we think of α as the value read off of the initial input stream, resulting in the final input stream and output stream
- In case of printing output, α would be an empty value
- Example of reading input
  `i = "def", o = "cba"`
  `i' = "ef", o' = "cba",` α `= "d"`
- As described earlier, it is in this way that we pass around the state of the world as a variable, in a referentially transparent manner.

The State Monad

- Thus a $f \in T(A)$ assigns to every possible initial state of the world $(\mathtt{i}, \mathtt{o})$ a final state $(\mathtt{i}', \mathtt{o}', \alpha)$
- We view f as the action of a single step of an algorithm
- In case we are reading input, we think of α as the value read off of the initial input stream, resulting in the final input stream and output stream
- In case of printing output, α would be an empty value
- Example of reading input
  `i = "def", o = "cba"`
  `i' = "ef", o' = "cba",` α `= "d"`
- As described earlier, it is in this way that we pass around the state of the world as a variable, in a referentially transparent manner.

Coding the State Monad

Given $f : A \to T(B)$ and $t \in T(A)$ we define `f >>> t` $\in T(B)$ to be

$$x \mapsto f(\text{return-value}(t(x)))(\text{new-state}(t(x)))$$

where we think of

$$\begin{aligned} T(B) &= \mathrm{Hom}(S, S \times B) \\ &= \mathrm{Hom}(\text{initial-state}, \text{new-state} \times \text{return-value}) \end{aligned}$$

and $S := \mathcal{S} \times \mathcal{S}$.

Coding the State Monad

Given $f : A \to T(B)$ and $t \in T(A)$ we define `f >>> t` $\in T(B)$ to be

$$x \mapsto f(\text{return-value}(t(x)))(\text{new-state}(t(x)))$$

where we think of

$$\begin{aligned} T(B) &= \mathrm{Hom}(S, S \times B) \\ &= \mathrm{Hom}(\text{initial-state}, \text{new-state} \times \text{return-value}) \end{aligned}$$

and $S := \mathcal{S} \times \mathcal{S}$.

Coding the State Monad

Given $f : A \to T(B)$ and $t \in T(A)$ we define `f >>> t` $\in T(B)$ to be

$$x \mapsto f(\text{return-value}(t(x)))(\text{new-state}(t(x)))$$

where we think of

$$\begin{aligned} T(B) &= \mathrm{Hom}(S, S \times B) \\ &= \mathrm{Hom}(\text{initial-state}, \text{new-state} \times \text{return-value}) \end{aligned}$$

and $S := \mathcal{S} \times \mathcal{S}$.

Back to our three line program

Let us compare the two definitions of Quicksort:

```
q []            = []
q (x:xs)        = q [y|y<-xs,y<x] ++ [x] ++ q [y|y<-xs,y>=x]
```

```
function Quicksort( Array )
...
return Concatenate( Quicksort( LessThanPivotArray ),
                    Pivot,
                    Quicksort( GreaterThanPivotArray ) )
```

Back to our three line program

Let us now consider the function liftM

```
liftM           :: (Monad m) => (a -> b) -> m a -> m b
liftM f xs      = (\y -> return (f y)) >>> xs

... liftM q ...
```

- Recall from before that we have `f >>> xs` $:= \mu_A(L(f)(xs)) = \mu_A((f(y) : y \in xs)) = \mathtt{concat}(f(y) : y \in xs)$
- Now, $q : L(A) \to L(A)$ so that `liftM q` is the function `g(xs) = (\y -> return (q y)) >>> xs` where xs is a list
- *In other words, it is the function which takes a list of lists and sorts each of the sublists therein, returning the result in list which is the result of concatenating all of those sorted sublists!*

Back to our three line program

Let us now consider the function liftM

```
liftM           :: (Monad m) => (a -> b) -> m a -> m b
liftM f xs      = (\y -> return (f y)) >>> xs
```

```
... liftM q ...
```

- Recall from before that we have `f >>> xs` := $\mu_A(L(f)(xs)) = \mu_A((f(y) : y \in xs)) = \mathtt{concat}(f(y) : y \in xs)$
- Now, $q : L(A) \to L(A)$ so that `liftM q` is the function `g(xs) = (\y -> return (q y)) >>> xs` where xs is a list
- *In other words, it is the function which takes a list of lists and sorts each of the sublists therein, returning the result in list which is the result of concatenating all of those sorted sublists!*

Back to our three line program

Let us now consider the function liftM

```
liftM          :: (Monad m) => (a -> b) -> m a -> m b
liftM f xs     = (\y -> return (f y)) >>> xs

... liftM q ...
```

- Recall from before that we have `f >>> xs` := $\mu_A(L(f)(xs)) = \mu_A((f(y) : y \in xs)) = \texttt{concat}(f(y) : y \in xs)$
- Now, $q : L(A) \to L(A)$ so that `liftM q` is the function `g(xs) = (\y -> return (q y)) >>> xs` where xs is a list
- *In other words, it is the function which takes a list of lists and sorts each of the sublists therein, returning the result in list which is the result of concatenating all of those sorted sublists!*

Back to our three line program

```
getArgs        :: State [String]
print          :: a -> State ()

main = print >>> (liftM q) getArgs
```

- We know that `getArgs` is of type `State [String]`
- thus it is an assigment which takes the initial input/output state and gives us the final input/output state as well as a list of strings read off of the input
- After what we have seen on `liftM q`, we know that it simply sorts the list of strings (and strips off a layer of parentheses)
- Finally, given the initial input/output state, print appends the resulting sorted list to the output and returns an empty value

Back to our three line program

```
getArgs       :: State [String]
print         :: a -> State ()

main = print >>> (liftM q) getArgs
```

- We know that `getArgs` is of type `State [String]`
- thus it is an assigment which takes the initial input/output state and gives us the final input/output state as well as a list of strings read off of the input
- After what we have seen on `liftM q`, we know that it simply sorts the list of strings (and strips off a layer of parentheses)
- Finally, given the initial input/output state, print appends the resulting sorted list to the output and returns an empty value

Back to our three line program

```
getArgs       :: State [String]
print         :: a -> State ()

main = print >>> (liftM q) getArgs
```

- We know that `getArgs` is of type `State [String]`
- thus it is an assigment which takes the initial input/output state and gives us the final input/output state as well as a list of strings read off of the input
- After what we have seen on `liftM q`, we know that it simply sorts the list of strings (and strips off a layer of parentheses)
- Finally, given the initial input/output state, print appends the resulting sorted list to the output and returns an empty value

Back to our three line program

```
getArgs       :: State [String]
print         :: a -> State ()

main = print >>> (liftM q) getArgs
```

- We know that `getArgs` is of type `State [String]`
- thus it is an assigment which takes the initial input/output state and gives us the final input/output state as well as a list of strings read off of the input
- After what we have seen on `liftM q`, we know that it simply sorts the list of strings (and strips off a layer of parentheses)
- Finally, given the initial input/output state, print appends the resulting sorted list to the output and returns an empty value

This is our program, in its entirety.

```
import System.Environment

(>>>)             :: (Monad m) => (a -> m b) -> m a -> m b
f >>> x           = x >>= f

liftM             :: (Monad m) => (a -> b) -> m a -> m b
liftM f t         = (\y -> return (f y)) >>> t

q                 :: (Ord a) => [a]->[a]
q []              = []
q (x:xs)          = q [y|y<-xs,y<x] ++ [x] ++ q [y|y<-xs,y>=x]

main = print >>> (liftM q) getArgs
```

```
> runghc quicksort.hs f g e d c a b
["a","b","c","d","e","f","g"]
```

Thank you.

```
-- the true type signatures
getArgs      :: IO [String]
print        :: (Show a) => a -> IO ()
```