Haskell and Domain-Specific Languages

Haskell nejen pro informatiky

Otakar Smrž

Institute of Formal and Applied Linguistics
Faculty of Mathematics and Physics
Charles University in Prague
otakar.smrz@mff.cuni.cz

https://wiki.ufal.ms.mff.cuni.cz/courses:pfl080
Part I

Higher-Order Functions
One of the most fancy higher-order functions is function composition:

\[(.) :: (a \to b) \to (c \to a) \to (c \to b)\]

\[(f \cdot g) x = f (g x)\]
Function Composition

One of the most fancy higher-order functions is function composition:

```
(·) :: (a → b) → (c → a) → c → b
(·) f g x = f (g x)
```
Function Composition

One of the most fancy higher-order functions is function composition:

\[( . ) :: (a \rightarrow b) \rightarrow (c \rightarrow a) \rightarrow (c \rightarrow b)\]
\[(f \ . \ g) \ x = f \ (g \ x)\]

\[\ldots\text{which is unlike function application:}\]

\[(\$) :: (a \rightarrow b) \rightarrow a \rightarrow b\]
\[f \ \$ \ x = f \ x\]
Function Composition

One of the most fancy higher-order functions is function composition:

\[(.) :: (a \rightarrow b) \rightarrow (c \rightarrow a) \rightarrow (c \rightarrow b)\]

\[(f \ . \ g)\ x = f (g\ x)\]

\[\ldots\text{which is unlike function application:}\]

\[($) :: (a \rightarrow b) \rightarrow (a \rightarrow b)\]

\[($)\ f = f\]
Function Composition

One of the most fancy higher-order functions is function composition:

\[(.) :: (a \rightarrow b) \rightarrow (c \rightarrow a) \rightarrow (c \rightarrow b)\]

\[(f \cdot g) x = f (g \ x)\]

... which is unlike function application:

\[(\$) :: (a \rightarrow b) \rightarrow (a \rightarrow b)\]

\[(\$) f = f\]

Recall fixity declarations, sections and the \``\ and (\) notations.
Function Composition

One of the most fancy higher-order functions is function composition:

\[(.): (a \rightarrow b) \rightarrow (c \rightarrow a) \rightarrow (c \rightarrow b)\]
\[(f \circ g) x = f (g x)\]

... which is unlike function application:

\[($) : (a \rightarrow b) \rightarrow (a \rightarrow b)\]
\[(f) f = f\]

Recall fixity declarations, sections and the ``` and () notations.

Visit Wikipedia for the explanation of η-conversion, β-reduction, and α-conversion in the lambda calculus.
Functions on Functions

Higher-order functions are those functions that take other functions as arguments.
Functions on Functions

Higher-order functions are those functions that take other functions as arguments.

Explore the Hugs definitions of the following functions:

- 
- (.), ($), flip, const, id
Functions on Functions

Higher-order functions are those functions that take other functions as arguments.

Explore the Hugs definitions of the following functions:

- `(.)`, `($)`, `flip`, `const`, `id`
- `curry`, `uncurry`, `iterate`, `until`
Higher-order functions are those functions that take other functions as arguments.

Explore the Hugs definitions of the following functions:

- (.), ($), flip, const, id
- curry, uncurry, iterate, until
- map, filter, zipWith
Higher-order functions are those functions that take other functions as arguments.

Explore the Hugs definitions of the following functions:

- (.), ($), flip, const, id
- curry, uncurry, iterate, until
- map, filter, zipWith
- foldr, foldl, foldr1, foldl1, scanr, scanl, unfoldr, unzip
Higher-order functions are those functions that take other functions as arguments.

Explore the Hugs definitions of the following functions:

- `(.)`, `(\$)`, `flip`, `const`, `id`
- `curry`, `uncurry`, `iterate`, `until`
- `map`, `filter`, `zipWith`
- `foldr`, `foldl`, `foldr1`, `foldl1`, `scanr`, `scanl`, `unfoldr`, `unzip`
- `takeWhile`, `dropWhile`, `span`, `break`, `groupBy`, `nubBy`
Functions on Functions

Higher-order functions are those functions that take other functions as arguments.

Explore the Hugs definitions of the following functions:

- $(.)$, $(\$)$, flip, const, id
- curry, uncurry, iterate, until
- map, filter, zipWith
- foldr, foldl, foldr1, foldl1, scanr, scanl, unfoldr, unzip
- takeWhile, dropWhile, span, break, groupBy, nubBy
- and, or, all, any, maximum, minimum
Functions on Functions

Higher-order functions are those functions that take other functions as arguments.

Explore the Hugs definitions of the following functions:

- (.), ($), flip, const, id
- curry, uncurry, iterate, until
- map, filter, zipWith
- foldr, foldl, foldr1, foldl1, scanr, scanl, unfoldr, unzip
- takeWhile, dropWhile, span, break, groupBy, nubBy
- and, or, all, any, maximum, minimum
- concat, length, elem, notElem, reverse
Higher-order functions are those functions that take other functions as arguments.

Explore the Hugs definitions of the following functions:

- $(.), \$(.), \text{flip}, \text{const}, \text{id}$
- $\text{curry}, \text{uncurry}, \text{iterate}, \text{until}$
- $\text{map}, \text{filter}, \text{zipWith}$
- $\text{foldr}, \text{foldl}, \text{foldr1}, \text{foldl1}, \text{scanr}, \text{scanl}, \text{unfoldr}, \text{unzip}$
- $\text{takeWhile}, \text{dropWhile}, \text{span}, \text{break}, \text{groupBy}, \text{nubBy}$
- $\text{and}, \text{or}, \text{all}, \text{any}, \text{maximum}, \text{minimum}$
- $\text{concat}, \text{length}, \text{elem}, \text{notElem}, \text{reverse}$
- $\text{product}, \text{sum}, \text{foldl'}$, ($\$!$)
Correctness

We can use **equational reasoning** to prove functions’ **properties**.
Correctness

We can use *equational reasoning* to prove functions’ properties.

\[
\text{flip} \cdot \text{flip} \simeq \text{id}
\]
Correctness

We can use **equational reasoning** to prove functions’ properties.

\[ \text{flip} \cdot \text{flip} \equiv \text{id} \]

\[ \text{map } f \cdot \text{map } g \equiv \text{map } (f \cdot g) \]
We can use equational reasoning to prove functions’ properties.

\[
\text{flip} \cdot \text{flip} \equiv \text{id}
\]

\[
\text{map } f \cdot \text{map } g \equiv \text{map } (f \cdot g)
\]

\[
\text{notElem} \equiv (\cdot) \text{not} \cdot \text{elem}
\]
We can use equational reasoning to prove functions’ properties.

\[
\text{flip} \cdot \text{flip} \equiv \text{id}
\]

\[
\text{map } f \cdot \text{map } g \equiv \text{map } (f \cdot g)
\]

\[
\text{notElem} \equiv (.) \text{ not} \cdot \text{elem}
\]

\[
\text{reverse} \equiv \text{foldl} (\text{flip } (:)) []
\]
Correctness

We can use **equational reasoning** to prove functions’ properties.

\[
\text{flip} \cdot \text{flip} \cong \text{id}
\]

\[
\text{map } f \cdot \text{map } g \equiv \text{map } (f \cdot g)
\]

\[
\text{notElem} \equiv (.) \text{not} \cdot \text{elem}
\]

\[
\text{reverse} \equiv \text{foldl} \ (\text{flip} \ ()): \ []
\]

Why did we write \(\cong\) somewhere, and not \(\equiv\) everywhere?
We can use equational reasoning to prove functions’ properties.

\[ \text{flip} \cdot \text{flip} \ \cong \ id \quad \quad \quad \quad (\text{flip} \cdot \text{flip}) \ f \ \equiv \ f \]

\[ \text{map} \ f \cdot \text{map} \ g \ \equiv \ \text{map} \ (f \cdot g) \]

\[ \text{notElem} \ \equiv \ (. \) \ \text{not} \ . \ \text{elem} \]

\[ \text{reverse} \ \equiv \ \text{foldl} \ (\text{flip} \ (\cdot)) \ [] \]

Why did we write \( \cong \) somewhere, and not \( \equiv \) everywhere?
Part II

Tree Structures
Write functions for **folding** and **linearizing** trees of these **types**:
Write functions for folding and linearizing trees of these types:

```haskell
data Tree a = Node a [Tree a]
```

Colored red-black trees can implement sets and finite maps. Study the zipper representation of trees by Huet [1].
Write functions for **folding** and **linearizing** trees of these **types**:

```haskell
data Tree a = Node a [Tree a]
```

```haskell
data BinTree a b = Fork a (BinTree a b) (BinTree a b) | Leaf b
```
Write functions for folding and linearizing trees of these types:

```haskell
data Tree a = Node a [Tree a]
```

```haskell
data BinTree a b = Fork a (BinTree a b) (BinTree a b) | Leaf b
```

Colored red-black trees can implement sets and finite maps.
Write functions for **folding** and **linearizing** trees of these **types**:

```haskell
data Tree a = Node a [Tree a]
```

```haskell
data BinTree a b = Fork a (BinTree a b) (BinTree a b) | Leaf b
```

Colored **red-black trees** can implement **sets** and **finite maps**.

Study the **zipper** representation of **trees** by Huet (1).
HaXmI is a library for processing XML and DTD that provides interesting DSL for document transformations.
HaXml is a library for processing XML and DTD that provides interesting DSL for document transformations.

```haskell
data Element = Elem Name [Attribute] [Content]
data Content = CElem Element
               | CText String
```

Read the paper by Wallace and Runciman [2] and try out Text.XML.HaXml.
HaXml is a library for processing XML and DTD that provides interesting DSL for document transformations.

data Element = Elem Name [Attribute] [Content]
data Content = CElem Element
   | CText String

type CFilter = Content -> [Content]
HaXml is a library for processing XML and DTD that provides interesting DSL for document transformations.

```haskell
data Element = Elem Name [Attribute] [Content]
data Content = CElem Element
               | CText String

type CFilter = Content -> [Content]
```

Read the paper by Wallace and Runciman (2) and try out Text.XML.HaXml.
Gérard Huet.
Functional Pearl. The Zipper.

Malcolm Wallace and Colin Runciman.
Haskell and XML: Generic combinators or type-based translation?