Towards internet of code

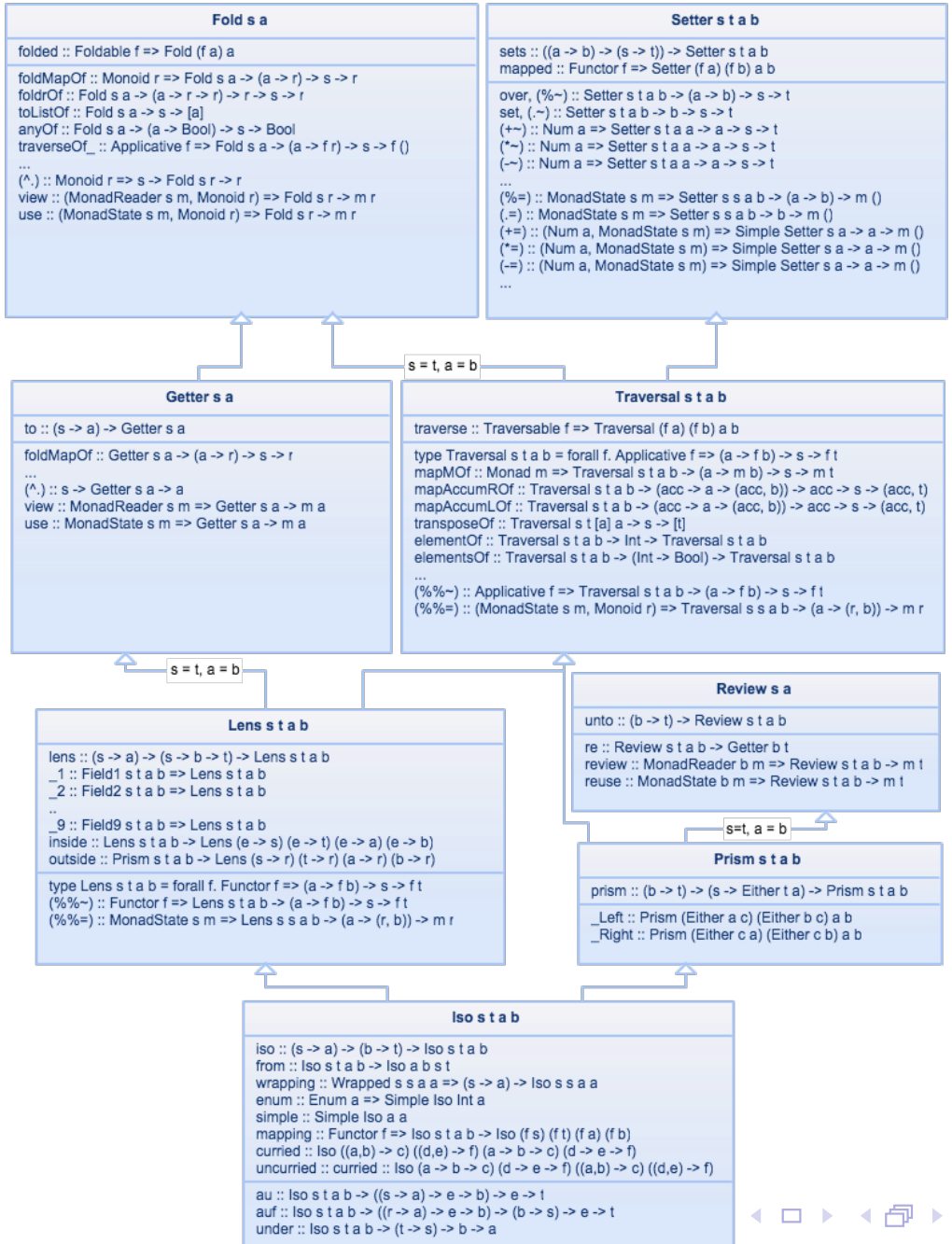
Łukasz Dąbek

May 27, 2015

This was supposed to be about `lens` library!

Yes, but...

This was supposed to be about `lens` library!



Why internet is awesome?

Because every device connected to it speaks the same language. This is what enabled its fast growth.

Why programming is awesome?

Because every programmer uses different language, libraries, data representations, all incompatible witch each other.

Why programming is awesome?

Because every programmer uses different language, libraries, data representations, all incompatible witch each other.

Take any useful piece of code and chances are that the same functionality is reimplemented in many languages.

The problem

Suppose we are writing a website in Haskell. Example: customized gift store.

The problem

Suppose we are writing a website in Haskell. Example: customized gift store.

```
price :: Product -> [Addons] -> Address -> Money
```

The problem

Suppose we are writing a website in Haskell. Example: customized gift store.

```
price :: Product -> [Addons] -> Address -> Money
```

```
function price(product, addons, address) { ... }
```

The problem

Client needs to know total cost of order. It is not just a sum of prices!

- Many shipping options $\Longrightarrow$ different prices.
- Shipping discount for big orders.
- *Buy two Combulbulators and get third FOR FREE!*

Non-solutions

- Make a request to server after each state change

Non-solutions

- Make a request to server after each state change (kills performance).

Non-solutions

- Make a request to server after each state change (kills performance).
- Implement logic in Haskell and JavaScript

Non-solutions

- Make a request to server after each state change (kills performance).
- Implement logic in Haskell and JavaScript (maintenance nightmare).

Non-solutions

- Make a request to server after each state change (kills performance).
- Implement logic in Haskell and JavaScript (maintenance nightmare).
- Write in JavaScript on the servers

Non-solutions

- Make a request to server after each state change (kills performance).
- Implement logic in Haskell and JavaScript (maintenance nightmare).
- Write in JavaScript on the servers (don't even get me started on this).

Non-solutions

- Make a request to server after each state change (kills performance).
- Implement logic in Haskell and JavaScript (maintenance nightmare).
- Write in JavaScript on the servers (don't even get me started on this).
- Embedding Haskell interpreter in JavaScript

Non-solutions

- Make a request to server after each state change (kills performance).
- Implement logic in Haskell and JavaScript (maintenance nightmare).
- Write in JavaScript on the servers (don't even get me started on this).
- Embedding Haskell interpreter in JavaScript (cumbersome).

Non-solutions

- Make a request to server after each state change (kills performance).
- Implement logic in Haskell and JavaScript (maintenance nightmare).
- Write in JavaScript on the servers (don't even get me started on this).
- Embedding Haskell interpreter in JavaScript (cumbersome).
- Use language compiled to native code and JavaScript (think `js_of_ocaml`)

Non-solutions

- Make a request to server after each state change (kills performance).
- Implement logic in Haskell and JavaScript (maintenance nightmare).
- Write in JavaScript on the servers (don't even get me started on this).
- Embedding Haskell interpreter in JavaScript (cumbersome).
- Use language compiled to native code and JavaScript (think `js_of_ocaml`) (not a bad solution really!).

Core of the problem

We have *shared logic*, operating on *shared data structures*.

In most use cases the functions implementing shared logic are pure.

The problem #2

We are writing social media client for Android and iOS. User interface code is completely separate, but code for data fetching should be almost the same.

The problem #2

We are writing social media client for Android and iOS. User interface code is completely separate, but code for data fetching should be almost the same.

Sometimes we are interested in sharing little more than pure functions.

The idea

Small language as a target for compilation *and decompilation*. Think of high level assembly language.

The idea

Small language as a target for compilation *and decompilation*. Think of high level assembly language.

It should be:

- pure,
- functional,
- simple, but expressive,
- typed, maybe even dependently typed.

Integration

From practical point of view interaction with shared language should be hassle free. This is wrong:

```
var ctx = new Morte.Context();
ctx.loadFile(...);
ctx.callFunction("price", Morte.ADT.List(...), ...);
```

This is better:

```
import '/my/awesome/library/prices';
price([product1], [], shipping_address);
```

This is better:

```
import '/my/awesome/library/prices';
price([product1], [], shipping_address);
```

After importing code it should be indistinguishable from JavaScript code in use.

In practice the system will do more interesting things, like mapping data types to representation idiomatic in host language. We will talk about this at the end.

The language

We will construct desired language. Let's start with simply typed lambda calculus:

$$E = x \mid \lambda(x : T).E \mid E\ E$$
$$T = X \mid T \to T$$

The language

We will construct desired language. Let's start with simply typed lambda calculus:

$$E = x \mid \lambda(x : T).E \mid E\ E$$
$$T = X \mid T \to T$$

I will use another syntax: `(fun (x:A) => x) y`

The language

We will construct desired language. Let's start with simply typed lambda calculus:

$$E = x \mid \lambda(x : T).E \mid E\ E$$
$$T = X \mid T \to T$$

I will use another syntax: `(fun (x:A) => x) y`

What can we express in this language?

Booleans

```
true = fun (x:A) (b:A) => x

false = fun (x:A) (b:A) => y
```

Booleans

```
true = fun (x:A) (b:A) => x

false = fun (x:A) (b:A) => y

if b x y = b x y
```

Natural numbers

```
zero = fun (f:A -> A) (z:A) => z

one = fun (f:A -> A) (z:A) => f z
```

Natural numbers

```
zero = fun (f:A -> A) (z:A) => z

one = fun (f:A -> A) (z:A) => f z

succ n = fun (f:A -> A) (z:A) => f (n f z)
```

Natural numbers

```
zero = fun (f:A -> A) (z:A) => z

one = fun (f:A -> A) (z:A) => f z

succ n = fun (f:A -> A) (z:A) => f (n f z)

add n m = fun (f:A -> A) (z:A) => n f (m f z)

mul n m = fun (f:A -> A) (z:A) => n (m f) z
```

Does it look like a `fold`?

STLC – problem

How to express identity function?

`fun (x:A) => x` is not polymorphic! We need richer type system.

System F

Polymorphic lambda calculus. We can quantify over types:

`id = fun (A:*) (x:A) => x` - polymorphic identity.

System F

Polymorphic lambda calculus. We can quantify over types:

`id = fun (A:*) (x:A) => x` - polymorphic identity.

The type of identity function is:
`id : forall (A:*). A -> A`

Pairs

```
Pair A B = forall (R:*). (A -> B -> R) -> R
```

```
fst : forall (A B:*).
      (forall (R:*). (A -> B -> R) -> R) -> A
```

In pseudonotation: `fst: forall (A B:*). Pair A B -> A`.

Pairs

```
Pair A B = forall (R:*). (A -> B -> R) -> R
```

```
fst : forall (A B:*).
      (forall (R:*). (A -> B -> R) -> R) -> A
```

In pseudonotation: `fst: forall (A B:*). Pair A B -> A.`

```
fst A B p = p A (fun (x:A) (y:B) => x)
snd A B p = p B (fun (x:A) (y:B) => y)
```

Natural number, honestly

```
Nat = forall (R:*). R -> (R -> R) -> R
```

Implementation of common functions are same as in STLC.

Lists

```
List A = forall (R:*). R -> (A -> R -> R) -> R

nil : forall (A:*). List A
cons : forall (A:*). A -> List A -> List A
map : forall(A B:*). (A -> B) -> List A -> List B
```

Lists

```
List A = forall (R:*). R -> (A -> R -> R) -> R

nil : forall (A:*). List A
cons : forall (A:*). A -> List A -> List A
map : forall(A B:*). (A -> B) -> List A -> List B


map A B f xs = xs (List B) (nil B xs)
    (fun (x:A) (ys:List B) => cons B (f x) ys)
```

We can represent algebraic data types in System F.

We can represent algebraic data types in System F. What else can we do?

Existential types

Suppose that we have a module with hidden type `S` and functions `f : S -> S`, `g : S -> Nat` and constant `c : S`. How to express it in System F?

Existential types

Suppose that we have a module with hidden type `S` and functions `f : S -> S`, `g : S -> Nat` and constant `c : S`. How to express it in System F?

```
forall (R:*).
  (forall (S:*). S -> (S -> S) -> (S -> Nat) -> R) -> R
```

Existential types

Suppose that we have a module with hidden type `S` and functions `f : S -> S`, `g : S -> Nat` and constant `c : S`. How to express it in System F?

```
forall (R:*).
  (forall (S:*). S -> (S -> S) -> (S -> Nat) -> R) -> R
```

System Fω

To get rid of pseudonotation for polymorphic list we need another, richer type system called Fω.

In a nutshell: we are introducing higher kinded types, also known as type constructors.

Lists, honestly

```
List : * -> *

List = fun (A:*) => forall (R:* -> *).
         R A -> (A -> R A -> R A) -> R A
```

Nothing else changed much.

Calculus of Constructions

Dependently typed version of Fω, basis for Coq (Calculus of Inductive Constructions).

Strong normalization

All of the mentioned languages are strongly normalizing.

What about the Android/iOS problem?

Possibly infinite behaviors

We can use streams for that! That was one of the first versions of Haskell I/O.

Possibly infinite behaviors

We can use streams for that! That was one of the first versions of Haskell I/O.

And because of strong normalization property we have progress guarantee for free.

If we have time left, we shall take a look at free monads.

Implementation – Morte and Annah

Morte is core language – currently something between CoC and System Fω.

Implementation – Morte and Annah

Morte is core language – currently something between CoC and System Fω.

Annah is higher level language compiled to *and from* Morte. Imports over network works now.

Most interesting feature – translating Morte data definitions into inductive definitions, GADT style.

Implementation – Morte and Annah

Morte is core language – currently something between CoC and System Fω.

Annah is higher level language compiled to *and from* Morte. Imports over network works now.

Most interesting feature – translating Morte data definitions into inductive definitions, GADT style.

Work in progress – decompilation of Annah to Haskell.

Implementation – Morte and Annah

The author is Garbriel Gonzalez, author of „Haskell for all” blog.

You can check out his Github profile and dive into the code!

Other solutions? LLVM? asm.js? One language to rule them all?

Thank you. Any questions?