Combinational Logic Based Language with Algebraic types (clbla)

Witalis Domitrz witekdomitrz@gmail.com

Introduction

The language is inspired by the Lambda Calcalculus course. It is meant to be purely functional, lazy, point-free language with Haskell-inspired syntax. One could argue that Haskell is not the best language to base on, but this interpreter is written in Haskell and it is the determining factor for the inspiration.

Language syntax

The whole programme consists of

- extensions,
- imports,
- environment.

The interpreter will attempt to print the result of `main` function which should be in the programme's environment. If it is not there the interpreter will fail with an appropriate error message.

Other behaviors might be added in the future and might be enabled using extensions.

Extensions

Names of extensions must start with a capital letter and contain only letters, digits, underscores (`_`), apostrophes (`'`) and asterisks (`*`).

To use an extension write:

```
{# LANGUAGE <name of the extension> #}
```

replacing `<name of the extension>` with the name of the extension you want to use in the beginning of the file. To use multiple extensions declare the next extension after the previous ones. Name of the extension should start with a capital letter.

Note that the order of the extensions might be important.

Already known extensions

- `FOn` - extension enabling automatically implemented folds
- `NElim` - extension disabling automatically implemented eliminators

If someone would like to have even more fun during coding, then they might consider using those extensions and implement everything using folds.

The extensions that might be added in the future are described in the features to be considered section.

Imports

Names of the modules must start with a capital letter and contain only letters, digits, underscores (`_`), apostrophes (`'`) and asterisks (`*`).

To import a module, which is associated with a file `<file name>.clbla` write

```
import <file name>
```

after the (maybe empty) list of extensions.

This import adds all the types and from environment of the `<file name>.clbla` file and all functions defined in that file (note that to the environment of the current file. Note that it may cause covering of some functions. Covering of the types is illegal, while covering of constructors is legal only across two different modules, but not in one module. If two modules import type with the same name the import will not fail if and ONLY if the types have exactly the same definitions.

To use multiple imports write multiple `import <file name>` directives separated by a semicolon (`;`) or in a new line each.

Environments

An environment consists of:

- types definitions,
- functions declarations,
- functions definitions.

All declared functions should be defined. The declarations and definitions order does not matter. There should be at most one entity with a given name defined in one environment. If there is a variable visible in the environment and it is redefined in that environment than it will be covered.

Types

The language allows algebraic and functional types. To construct an functional type use `->`. For example `a -> a` is a type of the identity function.

Types definitions

Each type must have at least one constructor. To define type `A` with parameters `b c` with constructors `D` with parameters of types `c` and `A b c` and `E` with parameter `b` write:

```
data A b c = D c (A b c) | E b
```

Types constructors The constructors of types (`D` and `E` in the example above) are treated as a functions from their arguments to that type. For example:

```
D :: c -> A b c -> A b c
E :: b -> A b c
```

Infix constructors One can define an infix constructor. Its name must start with `:` (colon) and follow the rules described in infix operators section.

Automatically generated eliminators The `data A b c = D c (A b c) | E b` declaration will automatically generate eliminator and fold (to enable fold use an extension):

```
elimA :: (c -> A b c -> d) -> (b -> d) -> A b c -> d
foldA :: (c -> d -> d) -> (b -> d) -> A b c -> d
```

Note that:

- eliminator for a non-recursive type is also a fold.
- fold can be created with an eliminator in an efficient way. For example:

```
foldA = w `b` b elimA `b` s (b `b` c `b` (b b)) foldA
```

it is obvious that not every eliminator can be created with fold in an efficient way, it can be created with some slowdown. To demonstrated this result I have created `Nats` module using only folds.

Proof of the conjecture For the sake of the simplicity I will focus on this type `data A b c = D c (A b c) | E b`, although this proof if fully general.

First construct a type which will represent object of type `A b c` and of parameters of its constructor.

```
data A' b c = D' (A b c) c (A b c) | E' (A b c) b
```

Its fold has the type `foldA' :: (A b c -> c -> A b c -> d) -> (A b c -> b -> d) -> A' b c -> d`. First we can implement function `valueA'` (which takes the value of `A b c` element from `A' b c` element discarding constructor parameter):

```
valueA' :: A' b c -> A b c
valueA' = foldA' (k `b` k) k
```

Now we can implement a `historyA` function which creates `A' b c` element from `A b c`.

```
historyA :: A b c -> A' b c
historyA = foldA (c s valueA' `b` s (c c) (b D' `b` b' valueA' `b` D)) (s (c E') E)
```

And now we get the eliminator in quite straightforward way:

```
elimA :: (c -> A b c -> d) -> (b -> d) -> A b c -> d
elimA = c c historyA `b` b b `b` b' k `b` foldA' `b` k
```

One could evaluate the proof in `clbla` with `FOn` and `NElim` extensions or in `Haskell` by adding:

```
foldA :: (c -> d -> d) -> (b -> d) -> A b c -> d
foldA f x (D c a) = f c (foldA f x a)
foldA _ x (E b) = x b

foldA' :: (A b c -> c -> A b c -> d) -> (A b c -> b -> d) -> A' b c -> d
foldA' f x (D' a c a') = f a c a'
foldA' _ x (E' a b) = x a b

k :: a -> b -> a
k = const
s :: (a -> b -> c) -> (a -> b) -> a -> c
s = (<*>)
b :: (b -> c) -> (a -> b) -> a -> c
b = s (k s) k -- b = (.)
c :: (a -> b -> c) -> b -> a -> c
c = s (b b s) (k k) -- c = flip
b' :: (a -> b) -> (b -> c) -> a -> c
b' = c b -- b' = flip (.)
```

QED

Natural numbers example Consider a definition of natural numbers as in `Nats.clbla`:

```
data Nat :: S Nat | Zero
```

then if not disabled one could use the eliminator to create predecessor function (predecessor of 0 is assumed to be 0):

```
pred :: Nat -> Nat
pred = elimNat i Zero
```

Real-life example Consider a definition of a polymorphic list type:

```
data List a = Next a (List a) | Nil
```

Its eliminator, `foldList :: (a -> b -> b) -> b -> List a -> b` is known as `foldr` in the Haskell language. It does not generate the `foldl` function however it can be easily written as

```
foldListL :: (a -> b -> a) -> a -> List b -> a
foldListL = b c (b (c foldList i) (b (b (c b)) c))
```

using `b` and `c` combinators from the `STD` library.

Second example One might want to use `Bool` type. To do so simply declare:

```
data Bool = True | False
```

and

```
if :: Bool -> a -> a -> a
if = c (b c foldBool)
```

Functions

By functions I do understand all variables (including 0-arguments functions).

Functions declarations

To declare a function write:

```
<function name> :: <function type>
```

Functions definition

To define a function write:

```
<function name> = <expression> [where <environment>]
```

The `where <environment>` part is optional. Replace the `<environment>` with an environment definition. The `<expression>` will be evaluated in this environment.

Build in functions

There are 2 build in functions.

```
s :: (a -> b -> c) -> (a -> b) -> a -> c
k :: a -> b -> a
```

satisfying:

```
s x y z = x z (y z)
k x y = x
```

std library

An addition to the interpreter is a std library. With at least these functions:

```
i :: a -> a
i = s k k

w :: (a -> a -> b) -> a -> b
w = s s (k i)

b :: (b -> c) -> (a -> b) -> a -> c
b = s (k s) k

c :: (a -> b -> c) -> b -> a -> c
c = s (b b s) (k k)

b' :: (a -> b) -> (b -> c) -> a -> c
b' = c b
```

Expressions

The expression might be:

- <variable name> - the value of this expression is the value of this variable,
- let <environment> in <expression> - the value of this expression is the value of <expression> evaluated in the <environment> environment.
- <expression1> <expression2> - the value of this expression is the value of application value of the <expression1> to the value of <expression2>.

Infix operators

An infix operator is either an operator whose name is composed from these !#$%&*+/<=>?@^|-~: characters starting with one of these $?|&<>=:+-*/^!. characters (base infix operators) or an ordinary function (including type constructors) name surrounded by ‘ (a grave).

Name starting with : are reserved for infix type constructors. Other base infix operators can be defined as an ordinary functions by using ( ) (round brackets). For example:

```
(++) = concat
```

is a valid function declaration (assuming that concat exists in this environment).

All base infix operators can be used as ordinary functions when surrounded by ( ) (round brackets). For example:

```
doubleList = w (++)
```

The priority is based on the first letter of the infix operator. The list of the operators first characters sorted by the priority:

- $
- ?
- |

- &
- < or > or =
- :
- + or -
- * or /
- ^
- ! or .
- ‘ (functions surrounded by ‘)

All infix operators have right-to-left associativity by design (because constructors and function composition is more important than + and -). (Might be changed in the future). Note that it means that 7 - 3 - 1 means 7 - (3 - 1).

Features to be considered

Infix operators with user-defined priorities

Priorities might be set using special syntax. The syntax tree would be rebuild after parsing.

Infix operators with user-defined associativity

Associativity might be set using special syntax. The syntax tree would be rebuild after parsing.

Pathetic extension

Adding build-in lambda* that would transform the programme tree as follows:

- $\lambda^* x.F = KF$ if x is not a free variable in F,
- $\lambda^* x.x = I$,
- $\lambda^* x.FG = S(\lambda^* x.F)(\lambda^* x.G)$.

with syntax

```
lambda* <variable> <expression>
```

where this whole term is treated as an <expression> modified as described before.

Note that this feature has very low priority.

An extension with build-in integers

This extension would make the programmes execute much faster due to the existence of specialized, build in CPUs’ instructions. This might be the default option.