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

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.

Note that the order of the extensions might be important. The extensions are evaluated in the order in which they are enabled in the file.

There are no extensions yet. The extensions that might be added in the future are described in the features to be considered section.

Imports

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 variables from environment of the `<file name>.clbla` file to the environment of the current file if there was no such an module imported before and does nothing otherwise.

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

Environments

An environment consists of:

- types declarations,
- 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.

Build in type

There is exactly one build in type - `Char`. It is build in to make print the results to the screen in more readable way.

Types declarations

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

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

Automatically generated eliminators

This declaration will automatically generate and eliminator:

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

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>`.

Features to be considered

Infix operators

Infix operators:

- with priorities depending on the name (as in `OCaml`),
- with priorities specified by the programmer.

If added then the expression `a <*>` would be equivalent to `(<*>) a` where `(<*>)` is treated as an ordinary function.

`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 expressions with build-in integers

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