Lambda Cases (lcases)

Dimitris Saridakis

Contents

1 Introduction

Haskell is a delightful language. Yet, for some reason, it doesn't seem to have it's rightful place in terms of popularity in industry. Why is it so? Is it inherently hard to learn and therefore only the brave enough students and corporations dare to use it, or could it be that the syntax is perplexing to the amateur eye? It is my belief that with some syntax changes that give a greater familiarity to the new user, there would be no language more compelling than (the new) Haskell. In an attempt to achieve that familiarity, I present some new syntax, of which some is closer to the imperative/OOP style (to attract more already experienced programmers from these languages), some is closer to mathematics (in which most programmers should be experienced) and some is closer to natural language (in which we are all already experienced).

2 Language Description

An lcases program consists of a set of value, type and predicate definitions along with type theorems. The "main" value determines the program's behaviour. Constants and functions are all considered values and they have no real distinction other than the fact that functions have a function type and constants don't. Functions (just like "values") can be passed to other functions as arguments or can be returned as a result of other functions.

Program example: extended euclidean alogirthm

```
// type definitions

tuple_type PrevCoeffs
value (prev_prev, prev : Int, Int)

tuple_type GcdAndCoeffs
value (gcd, a, b : Int, Int, Int)

// algorithm

extended_euclidean: (Int, Int) -> GcdAndCoeffs
  = (init_a_coeffs, init_b_coeffs) ==> ee_recursion

init_a_coeffs, init_b_coeffs: all PrevCoeffs
  = (1, 0), (0, 1)

ee_recursion: (PrevCoeffs, PrevCoeffs, Int, Int) -> GcdAndCoeffs
  = (a_coeffs, b_coeffs, x, cases) ->
    0 -> (x, a_coeffs.prev_prev, b_coeffs.prev_prev)
    y ->
      ee_recursion(next <== a_coeffs, next <== b_coeffs, y, x ==> mod <== y)
      where
      next: PrevCoeffs -> PrevCoeffs
        = fields -> (prev, prev_prev - x ==> div <== y * prev)

// reading, printing and main

read_two_ints : (Int x Int)WithIO
  = print <== "Please give me 2 ints";
    get_line >>= split_words o> apply(from_string)to_all o> ints ->
    ints ==> length ==> cases ->
      2 -> ints ==> with_env
      ... -> io_error <== "You didn't give me 2 ints"

print_gcd_and_coeffs : GcdAndCoeffs -> (Empty)WithIO
  = fields -> print("Gcd: " + gcd + "\nCoefficients: a = " + a + ", b = " + b)

main : (Empty)WithIO
  = read_two_ints >>= ints ->
    extended_euclidean(ints.1st, ints.2nd) ==> print_gcd_and_coeffs
```

2.1 Values

2.1.1 Literals

Literals are the same as haskell

Examples	Type
1, 2, 17, 42, -100	Int
1.61, 2.71, 3.14, -1234.567	Real
’a’, ’b’, ’c’, ’x’, ’y’, ’z’, ’.’, ’,’, ’\n’	Char
[1, 2, 3], [’a’, ’b’, ’c’], [1.61, 2.71, 3.14]	ListOf(Int)s, ListOf(Char)s, ListOf(Real)s
”Hello World!”, ”What’s up, doc?”, ”Alrighty then!”	String

2.1.2 Identifiers

An identifier is a string of lower case letters or underscore.

Grammar

$\langle \mathit{identifier} \rangle$ ::= ([a-z_])*

2.1.3 Operators

Operator	Type	Description	Associativity
`==>`	`(A, A -> B) -> B`	Right function application	Left
`<==`	`(A -> B, A) -> B`	Left function application	Left
`o>`	`(A -> B, B -> C) -> (A -> C)`	Right function composition	Left
`<o`	`(B -> C, A -> B) -> (A -> C)`	Left function composition	Right
`^`	`(A)ToThe(B)Gives(C) => (A, B) -> C`	General exponentiation	Right
`*`	`(A)And(B)MultiplyTo(C) => (A, B) -> C`	General multiplication	Left
`/`	`(A)Divides(B)To(C) => (A, B) -> C`	General division	Left
+	`(A)And(B)AddTo(C) => (A, B) -> C`	General addition	Left
`-`	`(A)SubtractsFrom(B)To(C) => (B, A) -> C`	General subtraction	Left
= /=	`(A)HasEquality => (A, A) -> Bool`	Equality operators	None
`> < >= <=`	`(A)HasOrder => (A, A) -> Bool`	Order operators	None
`& |`	`(Bool, Bool) -> Bool`	Boolean operators	Left
`>>=`	`(E)IsAnEnvironment => (E(A), A -> E(B)) -> E(B)`	Monad bind	Left
`;`	`(E)IsAnEnvironment => (E(A), E(B)) -> E(B)`	Monad then	Left

2.1.4 Expressions

Examples

```
42
```

```
x

funny_identifier

[1, 2, 3]

"Hello world!"

1.61 * 2.71 + 3.14

a -> 17 * a + 42

(x, y, z) -> (x^2 + y^2 + z^2)^(1/2)

n==>(+ 1)==>(^2)==>(* 3)==>print

f(x, y, z) + g(1, 2, 3)
```

Description

The base of expressions, are literals and identifiers, those can be combined either with operators, or by normal function application with mathematical notation. Finally, on top of that there can be added one of more abstractions (parameters) in the beginning of the expressions with an arrow.

Grammar

2.1.5 Definitions

Examples

```
foo : Int
  = 42

val1, val2, val3 : Int, Bool, Char
  = 42, true, 'a'

int1, int2, int3 : all Int
  = 1, 2, 3

succ : Int -> Int
  = x -> x + 1

f : (Int, Int, Int) -> Int
  = (a, b, c) -> a + b * c
```

Description

To define a new value you give it a name, a type and an expression. It is possible to group value definitions by seperating the names, the types and the expressions with commas. It is also possible to use the keyword "all" to give the same type to all the values.

Grammar

$\langle$*value-definitions*$\rangle$ ::= $\langle$*identifiers*$\rangle$ '␣:␣' ($\langle$*types*$\rangle$ | 'all' $\langle$*type*$\rangle$) '\n␣␣=' $\langle$*value-expressions*$\rangle$

$\langle$*identifiers*$\rangle$ ::= $\langle$*identifier*$\rangle$ (',␣' $\langle$*identifier*$\rangle$)*

$\langle$*types*$\rangle$ ::= $\langle$*type*$\rangle$ (',␣' $\langle$*type*$\rangle$)*

$\langle$*value-expressions*$\rangle$ ::= $\langle$*value-expression*$\rangle$ (',␣' $\langle$*value-expression*$\rangle$)*

Abstractions

```
x -> body
(x, y, z) -> body
cases -> body
(x, cases, z) -> body
```

2.2 Types

2.2.1 Type expressions

Examples

```
Int

String -> String

Int x Int

Int x Int -> Real

A -> A

(A -> B, B -> C) -> (A -> C)

((A, A) -> A, A, ListOf(A)s) -> A

((B, A) -> B, B, ListOf(A)s) -> B

(T)HasStringRepresantion => T -> String
```

Description

Examples	Description
Int Char String	Base types
A -> A (A -> B, B -> C) -> (A -> C)	Polymorphic types. A, B, C ... are type variables

Differences from Haskell

lcases	haskell	difference description
A -> A	a -> a	Type variables for polymorphic types are

Grammar

⟨*type*⟩ ::= ⟨*type-application*⟩ | ⟨*product-type*⟩ | ⟨*function-type*⟩

⟨*type-application*⟩ ::= [⟨*types-in-paren*⟩] ⟨*type-identifier*⟩ (⟨*types-in-paren*⟩ ([A-Za-z])*)* [⟨*types-in-paren*⟩]

⟨*types-in-paren*⟩ ::= ‘(’ ⟨*type*⟩ (‘, ’ ⟨*type*⟩)* ‘)’

⟨*type-identifier*⟩ ::= [A-Z] ([A-Za-z])*

⟨*product-type*⟩ ::= ⟨*product-subtype*⟩ (‘␣x␣’ ⟨*product-subtype*⟩)+

⟨*product-subtype*⟩ ::= ‘(’ (⟨*function-type*⟩ | ⟨*product-type*⟩) ‘)’ | ⟨*type-application*⟩

⟨*function-type*⟩ ::= ⟨*input-types-expression*⟩ ‘␣->␣’ ⟨*one-type*⟩

⟨*input-types-expression*⟩ ::= ⟨*one-type*⟩ | ⟨*two-or-more-types-in-paren*⟩

⟨*one-type*⟩ ::= ⟨*type-application*⟩ | ⟨*product-type*⟩ | ‘(’ ⟨*function-type*⟩ ‘)’

⟨*two-or-more-types-in-paren*⟩ ::= ‘(’ ⟨*type*⟩ (‘, ’ ⟨*type*⟩)+ ‘)’

2.2.2 Tuple Types

Definition Examples

```
tuple_type Name
value (first_name, last_name) : String x String

tuple_type ClientInfo
value (name, age, nationality) : Name x Int x String

tuple_type Date
value (day, month, year) : Int x Int x Int

tuple_type (A)And(B)
value (a_value, b_value) : A x B

tuple_type (ExprT)WithPosition
value (expr, line, column) : ExprT x Int x Int
```

Usage Examples

```
giorgos_info: ClientInfo
  = (("Giorgos", "Papadopoulos"), 42, "Greek")
```

```
john_info: ClientInfo
  = (("John", "Doe"), 42, "American")

name_to_string: Name -> String
  = fields -> "First Name: " + first_name + "\nLast Name: " + last_name

print_name_and_nationality : ClientInfo -> (Empty)WithIO
  = fields -> print(name ==> name_to_string + "\nNationality: " + nationality)

print_error_in_expr : (SomeDefinedExprT)WithPosition -> (Empty)WithIO
  = ewp ->
    print(
      "Error in the expression:" + es +
      "\nAt the position: (" + ls + ", " + cs + ")"
    )
    where
    es, ls, cs : all String
      = ewp.expr==>to_string, ewp.line==>to_string, ewp.column==>to_string
```

Description

Tuple types group many values into a single value. They are specified by their name, the names of their subvalues and the types of their subvalues. They generate projection functions for all of their subvalues by using a '.' before the name of the subvalue. For example the ClientInfo type above generates the following functions:

```
.name : ClientInfo -> String
.age : ClientInfo -> Int
.nationality : ClientInfo -> String
```

These functions shall be named "postfix functions" as the can just be appended to their argument.

Definition Grammar

⟨*tuple-type-definition*⟩ ::=
    ‘`tuple_type␣`’ ⟨*type-application*⟩ ‘`\nvalue␣`’ ‘`(`’ ⟨*identifier*⟩ (‘`,␣`’ ⟨*identifier*⟩)* ‘`)`’ ‘`␣:␣`’ ⟨*product-type*⟩

2.2.3 Or Types

Examples

```
or_type Bool
values true | false

or_type Possibly(A)
values the_value:A | no_value

or_type ListOf(A)s
values non_empty:HeadAndTailOf(A)s | empty

tuple_type HeadAndTailOf(A)s
value (head : A, tail : ListOf(A)s)

is_empty : ListOf(A)s -> Bool
  = cases ->
    empty -> true
```

```
    non_empty:anything -> false

get_head : ListOf(A)s -> Possibly(A)
  = cases ->
    empty -> no_value
    non_empty:list -> the_value:list.head
```

Description

Values of an or_type are one of many cases that possibly have other values inside. The cases which have other values inside are followed by a semicolon and the type of the internal value. The same syntax can be used for matching that particular case in a fucntion using the "cases" syntax, with the difference that after the colon, we write the name given to the value inside. Or_types and basic types are the only types on which the "cases" syntax can be used. The cases of an or_type which have a value inside create functions. For example, the case "non_empty" of a list creates the function "non_empty:" for which we can say:

```
non_empty: : HeadAndTailOf(A)s -> ListOf(A)s
```

Similarly:

```
the_value: : A -> Possibly(A)
```

These functions shall be named "prefix functions" as they are prepended to their argument. For example:

```
head_and_tail : HeadAndTailOf(Int)s
  = (1, [2, 3, 4])

list : ListOf(Int)s
  = non_empty:head_and_tail
```

These functions can be used like any other function as arguments to other functions. For example:

```
heads_and_tails_to_lists : ListOf(HeadAndTailOf(A)s)s -> ListOf(ListOf(A)s)s
  = apply(non_empty:)to_each
```

Definition Grammar

⟨*or-type-definition*⟩ ::=
    'or_type␣' ⟨*type-application*⟩ '\nvalues␣' ⟨*identifier*⟩ [':' ⟨*type*⟩] ('␣|␣' ⟨*identifier*⟩ [':' ⟨*type*⟩])*

2.3 Type Logic

2.3.1 Type Predicate

2.3.2 Type Theorem

2.4 Grammar

2.4.1 Tokens

Keywords

```
cases use_fields tuple_type or_type
```

Value names

Type names

2.4.2 Core Grammar

Program

⟨*program*⟩ ::= (⟨*value-definitions*⟩ | ⟨*type-def*⟩)+

⟨*value-definitions*⟩ ::= ⟨*identifiers*⟩ ‘`␣:␣`’ (⟨*types*⟩ | ‘`all`’ ⟨*type*⟩) ‘`\n␣␣=`’ ⟨*value-expressions*⟩

⟨*identifiers*⟩ ::= ⟨*identifier*⟩ (‘`,␣`’ ⟨*identifier*⟩)*

⟨*types*⟩ ::= ⟨*type*⟩ (‘`,␣`’ ⟨*type*⟩)*

⟨*value-expressions*⟩ ::= ⟨*value-expression*⟩ (‘`,␣`’ ⟨*value-expression*⟩)*

Types

Value Expressions

⟨*value-expression*⟩ ::= [⟨*input-expr*⟩] ⟨*cases-or-where*⟩ | ⟨*op-expr*⟩

⟨*cases-or-where*⟩ ::= ⟨*cases-expr*⟩ | ⟨*where-expr*⟩

⟨*where-expr*⟩ ::= ‘`let`’ ⟨*spicy-nl*⟩ (⟨*value-definitions*⟩ ⟨*spicy-nls*⟩)+ ‘`in`’ ⟨*value-expression*⟩ ⟨*spicy-nl*⟩

⟨*cases-expr*⟩ ::= ‘`cases`’ (⟨*case*⟩)+ [⟨*default-case*⟩]

3 Parser implimentation

The parser was implemented using the parsec library.

3.1 AST Types

3.2 Parsers

4 Translation to Haskell

5 Running examples

6 Conclusion

7 To be removed or incorporated

Addition/Subtraction:

```
+ : (A)HasAddition => (A, A) -> A
- : (A)HasSubtraction => (A, A) -> A
```

Equality and ordering:

```
= : (A)HasEquality => (A, A) -> Bool
<= : (A)HasOrder => (A, A) -> Bool
>= : (A)HasOrder => (A, A) -> Bool
```

(fmap)`<inside>` — (W)IsAWrapper => (A -> B, W(A)) -> W(B) — Apply inside operator
(<*>)`<wrapped_inside>` — (W)IsAWrapper => (W(A -> B), W(A)) -> W(B) — Order operators

better as postfix functions

Examples in Haskell

```
data ClientInfo =
  ClientInfoC String Int String

data WithPosition a =
  WithPositionC a Int Int

data Pair a b =
  PairC a b
```

Examples in Haskell

```
{-# language LambdaCase #-}

data Bool =
  Ctrue | Cfalse

data Possibly a =
  Cwrapper a | Cnothing

data ListOf_s a =
  Cnon_empty (NonEmptyListOf_s a) | Cempty

data NonEmptyListOf_s a =
```

```
  CNonEmptyListOf_s a (ListOf_s a)

is_empty :: ListOf_s a -> Bool
is_empty = \case
  Cempty -> Ctrue
  Cnon_empty (CNonEmptyListOf_s head tail) -> Cfalse

get_head :: ListOf_s a -> Possibly a
get_head = \case
  Cempty -> Cnothing
  Cnon_empty (CNonEmptyListOf_s head tail) -> Cwrapper head
```

Examples in Haskell

```
foo :: Int
foo = 42

val1 :: Int
val1 = 42
val2 :: Bool
val2 = true
val3 :: Char
val3 = 'a'

int1 :: Int
int1 = 1
int2 :: Int
int2 = 2
int3 :: Int
int3 = 3

succ :: Int -> Int
succ = \x -> x + 1

f :: Int -> Int -> Int -> Int
f = \a b c -> a + b * c
```

Or Types the following have automatically generated functions:

```
is_case:
```