Lambda Cases

Dimitris Saridakis

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. 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). Ofcourse, I could not rewrite Haskell from scratch by my self, so I will try to keep the language as small and to the point as possible for now, as well as use as much as I can from Haskell itself, as the new language (Lambda Cases) is compiled to Haskell.

2 Language Description

2.1 Values

As with Haskell a program is a set of values, wih the "main" value determining the program's behaviour.

2.1.1 Value Expressions

Operator Expressions

Function Application operators:

```
==> : (A, A -> B) -> B
<== : (A -> B, A) -> B
```

Addition/Subtraction:

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

Generalized addition/subtraction:

```
++ : (A)And(B)AddTo(C) => (A, B) -> C
-- : (A)And(B)SubtractTo(C) => (A, B) -> C
```

Equality and ordering:

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

Generalized equality/ordering:

```
== : (A)And(B)AreComparable => (A, B) -> Bool
=<= : (A)And(B)AreComparable => (A, B) -> Bool
=>= : (A)And(B)AreComparable => (A, B) -> Bool
```

Function composition:

```
o>: (A -> B, B -> C) -> (A -> C)
<o: (B -> C, A -> B) -> (A -> C)
```

Abstractions

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

Specific Abstractions

2.1.2 Value 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.

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
```

2.2 Types

2.2.1 Or Types

Examples

```
or_type Bool
values true | false

or_type Possibly<==A
values wrapper<==(value: A) | nothing

or_type ListOf(A)s
values non_empty<==(value: NonEmptyListOf(A)s) | empty

tuple_type NonEmptyListOf(A)s
value (head: T, tail: ListOf(A)s)

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

get_head: ListOf(A)s -> Possibly<==A
  = cases ->
    empty -> nothing
    non_empty -> head==>wrapper
```

Description

Values of an Or Type are one of many cases that possibly have values of other types inside. Or Types together with Int and Char are the only types on which the ”cases” syntax can be used.

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
```

Autogenerated Functions

Or Types the following have automatically generated functions:

`is_case:`

2.2.2 Tuple Types

Examples

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

tuple_type ExprT==>WithPosition
value (expr: ExprT, line: Int, column: Int)

tuple_type (FirstT, SecondT)==>Pair
value (first: FirstT, second: SecondT)
```

Description

Tuple types group many values into a single value.

Examples in Haskell

```
data ClientInfo =
  ClientInfoC String Int String

data WithPosition a =
  WithPositionC a Int Int

data Pair a b =
  PairC a b
```

Autogenerated Functions

2.3 Type Logic

Type Predicate

Type Theorem

2.4 Grammar

2.4.1 Tokens

Keywords

`cases use_fields tuple_type or_type`

Value names

⟨*value-name*⟩ ::= ⟨*lower-case-letter*⟩ (⟨*lower-case-letter*⟩ | ‘`_`’)*

Type names

⟨*type-name*⟩ ::= ⟨*upper-case-letter*⟩ (⟨*upper-case-letter*⟩ | ⟨*lower-case-letter*⟩)*

2.4.2 Core Grammar

Program

⟨*program*⟩	::= (⟨*value-defs*⟩ | ⟨*type-def*⟩)+
⟨*value-defs*⟩	::= ⟨*value-names*⟩ ‘`:␣`’ (⟨*types*⟩ | ‘`all`’ ⟨*type*⟩) ‘`\n␣␣=`’ ⟨*value-exprs*⟩
⟨*value-names*⟩	::= ⟨*value-name*⟩ (‘`,␣`’ ⟨*value-name*⟩)*
⟨*types*⟩	::= ⟨*type*⟩ (‘`,␣`’ ⟨*type*⟩)*
⟨*value-exprs*⟩	::= ⟨*value-expr*⟩ (‘`,␣`’ ⟨*value-expr*⟩)*

Types

⟨*type*⟩	::= ⟨*func-type*⟩ | ⟨*prod-type*⟩ | ⟨*type-app*⟩
⟨*func-type*⟩	::= ⟨*input-types-expr*⟩ ‘`␣->␣`’ ⟨*output-type*⟩
⟨*prod-type*⟩	::= ⟨*prod-sub-type*⟩ (‘`␣x␣`’ ⟨*prod-sub-type*⟩)+
⟨*type-app*⟩	::= [⟨*t-inputs*⟩ ‘`==>`’] ⟨*type-name*⟩ [‘`<==`’ ⟨*t-inputs*⟩]

⟨*input-types-expr*⟩ ::= ⟨*many-ts-in-paren*⟩ | ⟨*one-type*⟩

⟨*output-type*⟩ ::= ⟨*prod-type*⟩ | ⟨*type-app*⟩

⟨*prod-sub-type*⟩ ::= ‘`(`’ (⟨*func-type*⟩ | ⟨*prod-type*⟩) ‘`)`’ | ⟨*type-app*⟩

⟨*one-type*⟩ ::= ‘`(`’ ⟨*func-type*⟩ ‘`)`’ | ⟨*prod-type*⟩ | ⟨*type-app*⟩

⟨*t-inputs*⟩ ::= ⟨*many-ts-in-paren*⟩ | ‘`(`’ ⟨*type*⟩ ‘`)`’ | ⟨*type-name*⟩

⟨*many-ts-in-paren*⟩ ::= ‘`(`’ ⟨*type*⟩ (‘`, `’ ⟨*type*⟩)+ ‘`)`’

Value Expressions

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

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

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

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

3 Parser implimentation

The parser was implemented using the parsec library.

4 Translation to Haskell

5 Running examples

6 Conclusion