Aurox language specification

Jakub Grobelny

22 maja 2019

1 Syntax

Character classification:

- <*whitespace*> — HT, LF, CR, SPACE
- <*digit*> — 0-9
- <*lowercase*> — underscore or any other lowercase Unicode characters [1]
- <*uppercase*> — any uppercase Unicode characters [2]
- <*special*> — '-', '+', '*', '/', '=', '>', '<', '.', '!', '@', '%', '^', ' ', '&', '$', '|'

Any character sequence beginning with character # ending with LF are *comments*.

⟨*operator*⟩ ::= ⟨*special*⟩
| ⟨*special*⟩ ⟨*operator*⟩

⟨*keyword*⟩ ::= **let** | **in** | **if** | **and** | **or** | **then**
| **match** | **else** | **with** | **type** | **import**
| **define** | **_** | **defop** | **end** | **case**

[1] All characters `X` which satisfy `char_type(X, lower)` predicate in SWI Prolog
[2] All characters `X` which satisfy `char_type(X, upper)` predicate in SWI Prolog

⟨*identifier*⟩ ::= ⟨*lowercase*⟩ ⟨*alphanum*⟩

⟨*type name*⟩ ::= ⟨*uppercase*⟩ ⟨*alphanum*⟩

⟨*alphanum*⟩ ::= ⟨*alphanum char*⟩ ⟨*alphanum*⟩ | ϵ

⟨*alphanum char*⟩ ::= ⟨*lowercase*⟩ | **?** | ⟨*digit*⟩

⟨*integer*⟩ ::= ⟨*digit*⟩ | ⟨*digit*⟩ ⟨*integer*⟩

⟨*float*⟩ ::= ⟨*integer*⟩ . ⟨*digit sequence*⟩ ⟨*exponent*⟩ **e**
| ⟨*integer*⟩ ⟨*expontent*⟩

⟨*digit sequence*⟩ ::= ⟨*digit*⟩ | ⟨*digit*⟩ ⟨*digit sequence*⟩

⟨*e*⟩ ::= **e** | **E**

⟨*exponent*⟩ ::= ⟨*e*⟩ **-** ⟨*integer*⟩
| ⟨*e*⟩ ⟨*integer*⟩

⟨*boolean*⟩ ::= **false** | **true**

⟨*string*⟩ ::= ¨ ⟨*char sequence*⟩ ¨ | ¨¨

⟨*char*⟩ ::= ´ ⟨*character*⟩ ´

⟨*char sequence*⟩ ::= ⟨*character*⟩ | ⟨*character*⟩ ⟨*char sequence*⟩

⟨*character*⟩ ::= Unicode | \\ | **b** | **n** | **f** | **a** | **r** | **t** | **0** | \¨ | \ ´

⟨*program*⟩ ::= ⟨*operator declaration*⟩ ⟨*program*⟩
| ⟨*import*⟩ ⟨*program*⟩
| ⟨*expression sequence*⟩ ⟨*program*⟩
| ⟨*definition*⟩ ⟨*program*⟩
| ϵ

⟨*operator declaration*⟩ ::= **defop** ⟨*operator*⟩ ⟨*integer*⟩ ⟨*associativity*⟩

⟨*associativity*⟩ ::= **left** | **right** | **none** | **prefix** | **postfix**

⟨*import*⟩ ::= **import** ⟨*import list*⟩ **end**

⟨*import list*⟩ ::= ϵ | ⟨*string*⟩ ⟨*import list*⟩
| ⟨*type name*⟩ ⟨*import list*⟩

⟨*definition*⟩ ::= **define** ⟨*function name*⟩ ⟨*formal parameters*⟩ **:**
⟨*type*⟩ = ⟨*expression sequence*⟩ **end**

⟨*function name*⟩ ::= ⟨*identifier*⟩ | **(** ⟨*operator*⟩ **)**

⟨*formal parameters*⟩ ::= ⟨*variable name*⟩ ⟨*formal parameters*⟩ | ϵ

⟨*variable name*⟩ ::= ⟨*identifier*⟩ | **_**

⟨*type*⟩ ::= ⟨*function type*⟩
| ⟨*function type*⟩ **,** ⟨*tupe*⟩

⟨*function type*⟩ ::= ⟨*algebraic data type*⟩
| ⟨*function type*⟩ **(->)** ⟨*algebraic data type*⟩

⟨*algebraic data type*⟩ ::= ⟨*type name*⟩ ⟨*atomic type sequence*⟩
| ⟨*atomic type*⟩

⟨*atomic type sequence*⟩ ::= ⟨*atomic type*⟩ ⟨*atomic type sequence*⟩ | ϵ

⟨*atomic type*⟩ ::= ⟨*identifier*⟩ | ⟨*type name*⟩
| **[** ⟨*type*⟩ **]** | **(** ⟨*type*⟩ **)**

⟨*type definition*⟩ ::= **type** ⟨*type name*⟩ ⟨*formal parameters*⟩ **with**
⟨*type constructors*⟩ **end**

⟨*type constructors*⟩ ::= **case** ⟨*type name*⟩ ⟨*atomic type*⟩
| **case** ⟨*type name*⟩

⟨*expression sequence*⟩ ::= ⟨*expression*⟩
| ⟨*expression*⟩ **;** ⟨*expression sequence*⟩

⟨*expression*⟩ ::= ⟨*pattern matching*⟩ | ⟨*let definition*⟩
| ⟨*conditional expression*⟩ | ⟨*tuple expression*⟩

⟨*let definition*⟩ ::= **let** ⟨*variable name*⟩ **:** ⟨*type*⟩ =
⟨*expression sequence*⟩ **in** ⟨*expression sequence*⟩ **end**

⟨*conditional expression*⟩ ::= **if** ⟨*expression sequence*⟩ **then**
⟨*expression sequence*⟩ **else** ⟨*expression sequence*⟩ **end**

⟨*pattern matching*⟩ ::= **match** ⟨*expression sequence*⟩ **with**
⟨*pattern matching cases*⟩ **end**

⟨*pattern matching cases*⟩ ::= ⟨*pattern case*⟩ ⟨*pattern matching cases*⟩
| ϵ

⟨*pattern case*⟩ ::= **case** ⟨*pattern*⟩ => ⟨*expression sequence*⟩

⟨*pattern*⟩ ::= ⟨*deconstructor pattern*⟩
| ⟨*deconstructor pattern*⟩ **,** ⟨*pattern*⟩

⟨*deconstructor pattern*⟩ ::= ⟨*type name*⟩ ⟨*atomic pattern*⟩
| ⟨*atomic pattern*⟩

⟨*atomic pattern*⟩ ::= ⟨*variable name*⟩ | ⟨*type name*⟩
| **(** ⟨*pattern*⟩ **)**
| ⟨*list pattern*⟩
| ⟨*constant*⟩

⟨*list pattern*⟩ ::= **[** ⟨*pattern*⟩ | ⟨*variable name*⟩ **]**
| **[** ⟨*pattern*⟩ **]**
| **[ ]**

⟨*constant*⟩ ::= ⟨*integer*⟩ | ⟨*boolean*⟩ | ⟨*float*⟩
| **( )** | ⟨*string*⟩ | ⟨*char*⟩

⟨*tuple expression*⟩ ::= ⟨*logical or*⟩ **,** ⟨*tuple expression*⟩
| ⟨*logical or*⟩

⟨*logical or*⟩ ::= ⟨*logical and*⟩ **and** ⟨*logical or*⟩
| ⟨*logical and*⟩

⟨*logical and*⟩ ::= ⟨*expression none 0*⟩ **and** ⟨*logical and*⟩
| ⟨*expression none 0*⟩

⟨*expression none N*⟩ ::= ⟨*expression right N*⟩ ⟨*operator none N*⟩
⟨*expression none N*⟩
| ⟨*expression right N*⟩

⟨*expression right N*⟩ ::= ⟨*expression left N*⟩ ⟨*operator right N*⟩
⟨*expression right N*⟩
| ⟨*expression left N*⟩

⟨*expression left N*⟩ ::= ⟨*expression postfix N*⟩ ⟨*operator left N*⟩
⟨*expression left N*⟩
| ⟨*expression postfix N*⟩

⟨*expression postfix N*⟩ ::= ⟨*expression prefix N*⟩ ⟨*operator postfix N*⟩
| ⟨*expression prefix N*⟩

⟨*expression prefix 20*⟩ ::= ⟨*operator prefix 20*⟩ ⟨*application*⟩
| ⟨*application*⟩

⟨*expression prefix N*⟩ ::= ⟨*operator prefix N*⟩ ⟨*expression none (N+1)*⟩
| ⟨*expression none (N+1)*⟩

⟨*application*⟩ ::= ⟨*atomic expression*⟩ ⟨*application*⟩
| ⟨*atomic expression*⟩

⟨*atomic expression*⟩ ::= ⟨*constant*⟩
| **(** ⟨*expression sequence*⟩ **)**
| ⟨*list expression*⟩
| ⟨*lambda expression*⟩
| **`** ⟨*operator*⟩
| ⟨*identifier*⟩
| ⟨*type name*⟩

⟨*lambda expression*⟩ ::= **{** | ⟨*formal parameters*⟩ | ⟨*expression sequence*⟩ **}**

⟨*list expression*⟩ ::= **[ ]**
| **[** ⟨*tuple expression*⟩ **]**
| **[** ⟨*tuple expression*⟩ | ⟨*logical or*⟩ **]**

2 Semantics

3 Type system

$$\frac{\Gamma \vdash e_1 :: \alpha \to \tau \quad \Gamma \vdash e_2 :: \alpha}{\Gamma \vdash e_1\, e_2 :: \tau}$$

$$\frac{\Gamma \vdash c :: Bool \quad \Gamma \vdash e_1 :: \tau \quad \Gamma \vdash e_2 :: \tau}{\Gamma \vdash if\, c\, then\, e_1\ else\, e_2\ end :: \tau}$$

$$\frac{}{\Gamma \vdash [\,] :: [\tau]}$$

$$\frac{\Gamma \vdash e_1 :: \tau \quad \Gamma \vdash [e_2, ..., e_n] :: [\tau]}{\Gamma \vdash [e_1, e_2, ...e_n] :: [\tau]}$$

$$\frac{\Gamma \vdash [e_1, e_2, ...e_n] :: [\tau] \quad \Gamma \vdash e_{n+1} :: [\tau]}{\Gamma \vdash [e_1, e_2, ...e_n \,|\, e_{n+1}] :: [\tau]}$$

$$\frac{\Gamma \vdash e_1 :: \tau_1 \quad \Gamma \vdash e_2 :: \tau_2}{\Gamma \vdash e_1, e_2 :: \tau_1, \tau_2}$$

$$\frac{\Gamma \vdash e_n :: \tau}{\Gamma \vdash e_1; e_2; ...e_n :: \tau}$$

$$\frac{\Gamma \vdash e_1 :: Bool \quad \Gamma \vdash e_2 :: Bool}{\Gamma \vdash e_1\, and\, e_2 :: Bool}$$

$$\frac{\Gamma \vdash e_1 :: Bool \quad \Gamma \vdash e_2 :: Bool}{\Gamma \vdash e_1\, or\, e_2 :: Bool}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x :: \tau}$$

$$\frac{}{\Gamma \vdash n :: Int}$$

$$\frac{}{\Gamma \vdash x :: Float}$$

$$\frac{}{\Gamma \vdash () :: Unit}$$

$$\frac{}{\Gamma \vdash true :: Bool}$$

$$\frac{}{\Gamma \vdash false :: Bool}$$

Jeżeli $\otimes$ jest operatorem binarnym, to wówczas

$$\frac{\Gamma \vdash \otimes :: \alpha \to \beta \to \tau \quad \Gamma \vdash e_1 :: \alpha \quad \Gamma \vdash e_2 :: \beta}{\Gamma \vdash e_1 \otimes e_2 :: \tau}$$

Jeżeli $\otimes$ jest prefiksowym operatorem unarnym, to wówczas

$$\frac{\Gamma \vdash \otimes :: \alpha \to \tau \quad \Gamma \vdash e :: \alpha}{\Gamma \vdash \otimes e :: \tau}$$

Jeżeli $\otimes$ jest postfiksowym operatorem unarnym, to wówczas

$$\frac{\Gamma \vdash \otimes :: \alpha \to \tau \quad \Gamma \vdash e :: \alpha}{\Gamma \vdash e \otimes :: \tau}$$