System F Language Specification

Syntax

e	:=	lit	literals
		eid	expression identifier
		`(`e`)`	parenthesized
		e`[`τ`]`	type application
		$e_1\ e_2$	application
		`(`$\overline{e}^{(,)\geq 2}$`)`	tuples
		$e_1\ op\ e_2$	infix operation
		$\Lambda\ \overline{tid}^{(,)\geq 1}.\ e$	parametric polymorphism
		$\lambda\ \overline{eid\ :\ \tau}^{(,)\geq 1}.\ e$	anonymous function
		`if` e_1 `then` e_2 `else` e_3	if expression
		`let` eid `:` τ `=` e_1 `in` e_2	let binding
		`match` e `with` $\overline{\texttt{|}\ pat\ \texttt{=>}\ e}^{+}$	pattern destructing
τ	:=	tid	type identifier
		`(`τ`)`	parenthesized
		τ_1 `*` τ_2	tuple types
		τ_1 `->` τ_2	arrow types
		$\forall\ \overline{tid}^{(,)\geq 1}.\ \tau$	universal types
		`Int` | `Bool` | `Unit`	built-in types
lit	:=	`null`	unit literal
		`true` | `false`	boolean literals
		... | $\sim$`2` | $\sim$`1` | `0` | `1` | `2` | ...	64 bit signed integers
pat	:=	`_`	wildcard, producing no bindings
		eid `:` τ	binds a single expression
		`(`$\overline{pat}^{(,)\geq 2}$`)`	destructs a tuple

Multiple argument λ's (and $\forall$'s) are syntactic sugar for nested functions. For instance,

$$\lambda\ \text{x: } \texttt{int}, \text{ y: } \texttt{int}.\ \text{x} + \text{y} \stackrel{\text{def}}{=} \lambda\ \text{x: } \texttt{int}.\ \lambda\ \text{y: } \texttt{int}.\ \text{x} + \text{y}$$

Semantics:

CBV big step semantics with capture-avoiding substitution.
When a bound variable is bound again, the new binding takes over.
There is no one-type tuples
Lexical scope.