Polylamb Language Specification

1 Syntax

1.1 Expressions

e	:=	l	literals
		x	variables (lowercase)
		`(`e`)`	parenthesized
		e`[`τ`]`	type application
		$e_1\ e_2$	application
		$e_1\ op\ e_2$	binary operation
		`(`e_1`, ...,` e_n`)`	n-tuples, $n \geq 2$
		`λ` x`:`τ`.` e	lambda abstraction[1]
		`Λ` t`.` e	type abstraction[2]
		`let` p `=` e_1 `in` e_2	let binding
		`fix` f `=` `λ (`x`:`τ`):`τ'`.` e_1 `in` e_2	recursive functions[3]
		`if` e_1 `then` e_2 `else` e_3	if expression
l	:=	`null`	unit literal: `Unit`
		`true` | `false`	boolean literals: `Bool`
		... | `−2` | `−1` | `0` | `1` | `2` | ...	64-bit signed ints: `Int`
p	:=	`_`	discarded pattern
		x	single argument
		`(`p_1`, ...,` p_n`)`	n-tuple destructor, $n \geq 2$
op	:=	`&` | `|` | `<` | `=` | `>` | `+` | `-` | `*`	binary operations

1.2 Types

τ	:=	t	type variable (uppercase)
		`(`τ`)`	parenthesized
		τ_1 `->` τ_2	arrow types
		`∀` t`.` τ	universal types
		τ_1 `*` ... `*` τ_n	tuple types, $n \geq 2$
		`Int` | `Bool` | `Unit`	built-in types

δ	:=	**let** p **=** e	declaration

Declarations

Alternative syntax

We can write \ or **lambda** instead of **λ**.
We can write **any** in place of **Λ**.
We can write **forall** in place of ∀.

Semantics:

Call-by-value big step semantics.
When a bound variable is bound again, the new binding takes over.
There is no one-type tuples
Lexical scope.

[1]One can also denote a curried multi-argument function with the syntax **λ (**v_1**:**t_1**)** ... **(**v_n**:**t_n**).** e, which desugars to n nested lambda expressions. Note that in this case, parentheses are needed around each annotated argument.

[2]Similarly, **Λ** t_1 ... t_n**.** e is n nested type abstractions.

[3]To define mutually recursive functions, connect the definitions using the keyword **and**. For example, one can have **fix** f_1 **= λ (**v_1**:**τ_1**) ->** τ_1'**.** e_1 **and ... and** f_n **= λ (**v_n**:**τ_n**) ->** τ_n'**.** e_n **in** e'