System F Language Specification

Syntax

Expressions

e	$:=$	l	literals
		v	expression identifier
		$\texttt{(}e\texttt{)}$	parenthesized
		$e\texttt{[}\tau\texttt{]}$	type concretization
		$e_1\ e_2$	application
		$e_1\ op\ e_2$	binary operation
		$\texttt{(}e_1,\ \ldots,\ e_n\texttt{)}$	n-tuples, $n \geq 2$
		$\boldsymbol{\lambda}\ v\texttt{:}\tau\texttt{.}\ e$	lambda abstraction[1]
		$\boldsymbol{\Lambda}\ t\texttt{.}\ e$	type abstraction[2]
		$\texttt{let}\ p\ \texttt{=}\ e_1\ \texttt{in}\ e_2$	let binding
		$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$	if expression
l	$:=$	`null`	unit literal: `Unit`
		`true` | `false`	boolean literals: `Bool`
		… | `−2` | `−1` | `0` | `1` | `2` | …	64-bit signed ints: `Int`
p	$:=$	$\texttt{\_}\ \texttt{:}\ \tau$	discarded pattern
		$v\ \texttt{:}\ \tau$	single argument
		$\texttt{(}p_1,\ \ldots,\ p_n\texttt{)}$	n-tuple destructor, $n \geq 2$

Types

τ	$:=$	t	type identifier
		$\texttt{(}\tau\texttt{)}$	parenthesized
		$\tau_1\ \texttt{->}\ \tau_2$	arrow types
		$\forall\ t\texttt{.}\ \tau$	universal types
		$\tau_1\ \texttt{*}\ \ldots\ \texttt{*}\ \tau_n$	tuple types, $n \geq 2$
		`Int` | `Bool` | `Unit`	built-in types

Declarations

δ	$:=$	$\texttt{let}\ p\ \texttt{=}\ e$	declaration

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

[2] Similarly, $\boldsymbol{\Lambda}\ t_1\ \ldots\ t_n\texttt{.}\ e$ is n nested type abstractions.

Alternative syntax

We can write \ or `lambda` instead of $\boldsymbol{\lambda}$.
We can write `any` in place of $\boldsymbol{\Lambda}$.
We can write `forall` in place of $\forall$.

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.