Function Combinators with Ramda

Original Idea comes from Avaq/combinators.js.

Name	#	Haskell	Ramda	Crocks	Functional Signature	`Functor m ⇒`	Function `f, g, h` Evaluation
identity	**I**	`id`	`identity`	`identity`	`a → a`		
constant	**K**	`const`	`always`	`constant`	`a → b → a`		
eager application[1]	**A**	`($)`	`call`		`(() → b) → b`		
lazy application			`thunkify`, `partial`		`(a → b) → a → (() → b)`		
thrush	**T**	`(&)`	`applyTo`	`applyTo`	`a → (a → b) → b`		
tap			`tap`	`tap`	`(a → b) → a → a`		
flip	**C**	`flip`	`flip`	`flip`	`(a → b → c) → b → a → c`		`f(b, a)`
compose	**B**	`(.)`, `fmap`[2]	`map`[2], `o`	`composeB`	`(b → c) → (a → b) → a → c`	`(b → c) → m b → m c`	`f(g(a))`
substitution	**S**	`ap`[2]	`ap`[2]	`substitution`	`(a → b → c) → (a → b) → a → c`	`m (b → c) → m b → m c`	`f(a, g(a))`
chain			`chain`[2]		`(a → b → c) → (b → a) → b → c`	`(a → m c) → m a → m c`	`f(g(b), b)`
duplication	**W**	`join`[2]	`unnest`[2]		`(a → a → b) → a → b`	`m (m b) → m b`	`f(a, a)`
lift			`converge`, `lift`	`converge`	`(b → c → d) → (a → b) → (a → c) → a → d`	`(b → c → d) → m b → m c → m d`	`f(g(a), h(a))`
useWith			`useWith`	`compose2`	`(c → d → e) → (a → c) → (b → d) → a → b → e`		`f(g(a), h(b))`
psi	**P**	`on`	`on`	`psi`	`(b → b → c) → (a → b) → a → a → c`		`f(g(a1), g(a2))`
"compose inner"			`map`[3]		`(a → d → c) → (b → d) → (a → b → c)`	`Functor n ⇒ m (d → c) → n d → m n c`	`f(a, g(b))`
"lift partial"			`map`[4]		`(d → b → c) → (a → d) → (a → b → c)`	`Functor n ⇒ (d → n c) → m d → m n c`	`f(g(a), b)`
unit				`unit`	`() → undefined`		

[1]) The A-combinator can be implemented as an alias of the I-combinator. Its implementation in Haskell exists because the infix nature gives it some utility. Its implementation in Ramda exists because it is overloaded with additional functionality.

[2]) Algebras like `ap` have different implementations for different types. They work like Function combinators only for Function inputs.

[3]) `(mfd2c, nd) => map( fd2c => map(fd2c)(nd) )( mfd2c )`

[4]) `(fd2nc, md) => map(fd2nc, md)`