1 Syntax

$\kappa ::=$		kinds
	Type	kind of values
	Row	kind of rows
	$\kappa \to \kappa$	kind arrow

$\pi ::=$		constraints
	τ / l	row lacks label
	value(τ)	type is not an effect type
	implicit(τ)	value of type is in enviroment

$\sigma ::=$		polytypes
	$\forall \bar{\alpha} \,.\, \bar{\pi} \Rightarrow \tau$	forall with constraints

$\tau ::=$		monotypes
	c	type constructor
	α	type variable
	$\tau \; \tau$	type application
	$<>$	empty row
	$< \mathrm{l} : \tau \mid \tau >$	row extension
	$\mu\alpha \,.\, \tau$	recursive type

$e ::=$		terms
	l	label
	x	variable
	$e\ e$	application
	$e\ \{e\}$	implicit application
	$\lambda x\ .\ e$	abstraction
	$\lambda\{x\}\ .\ e$	implicit abstraction
	let $x = e$ in e	let expression
	letr $x = e$ in e	recursive let expression
	$e : \tau$	type annotation
	$\{\}$	empty record
	$.l$	record selection
	$.+l$	record extension
	$.-l$	record restriction
	$.:l$	record update
	$@l$	variant injection
	$@+l$	variant embedding
	$@:l$	variant update
	$?l$	variant elimination
	end	end variant elimination
	$x \leftarrow e; e$	effect sequencing
	$e; e$	effect sequencing (discard argument)
	$!l$	perform effect
	$!+l$	embed effect
	handle { return $x \to x, l\ x\ k \to e, ...\}$	effect handling
	pure e	unwrap value from effect type
	return e	wrap value in effect type

1.1 Type constants

Rec : Row → Type
Var : Row → Type
Eff : Row → Type → Type

2 Examples

Listing 1: Records and variants

```
.x { x = 10 } == 10
.+x 10 {} == { x = 10 }
.-x { x = 10 } == {}
.:x (\x -> x + 1) { x = 10 } == { x = 11 }

@Just 10 == @Just 10
@:Just (\x -> x + 1) (@Just 10) == @Just 11
?Just (\x -> x + 1) (\_ -> 0) (@Just 10) == 1
?Just (\x -> x + 1) (\_ -> 0) (@Nothing {}) == 0
```

Listing 2: Basic effect handlers

```
// define flip action
flip : Eff { Flip : {} -> Bool | r } Bool
flip = !Flip {}

// program that uses the flip effect
program : Eff { Flip : {} -> Bool } Bool
program =
  x <- flip;
  y <- flip;
  return (x || y)

// handler that always returns True
alwaysTrue : Eff { Flip : {} -> Bool | r } t -> Eff r t
alwaysTrue = handle { Flip {} k -> k True }

// result of the program
result : Bool
result = pure (alwaysTrue program) == True
```

Listing 3: State effect

```
// state effect handler (v = initial state)
state :
  Eff { Get : {} -> v, Set : v -> {} | r } t ->
  Eff r (v -> Eff p t)
state = handle {
  Get _ k -> return (\v -> (f <- k v; f v))
  Set v k -> return (\_ -> (f <- k v; f v))
  return v -> return (\_ -> return v)
}

get = !Get {}
set v = !Set v

program =
  x <- get;
  set 10;
  y <- get;
  return (x + y)

result = pure (pure (state program) 100) == 110
```

Listing 4: IO effects

```
runIO :
  Eff {
    putLine : Str -> {},
    getLine : {} -> Str
    | r
  } t -> Eff r t

infiniteGreeter =
  name <- getLine;
  putLine ("Hello " ++ name ++ "!");
  infiniteGreeter

main = runIO infiniteGreeter
```

Listing 5: Recursion effect

```
// fix as an effectful function
fix : (t -> t) -> Eff { fix : (t -> t) -> t | r } t
runFix : Eff { fix : (t -> t) -> t | r} x -> Eff r x

facEff : Eff { fix : (Int -> Int) -> Int | r } (Int -> Int)
facEff = fix (\fac n ->
  if (n <= 1)
    1
  else
    n * (fac (n - 1)))

fac : Int -> Eff { fix : (Int -> Int) -> Int | r } Int
fac n =
  f <- fac;
  return (f n)

main = runFix (fac 10)
```

Listing 6: Implicits

```
Show : (t -> Str) -> Var { Show : t -> Str }
Show f = @Show f

show : Var { Show : t -> Str } -> (t -> Str)
show {showInstance} = (?Show id end) showInstance

int2string : Int -> Str

showInt : Var { Show : Int -> Str }
showInt = Show int2string

main : forall {} . implicit(Var { Show : Int -> Str }) => Str
main = (show 10) ++ (show {showInt} 10)
```

3 Papers

3.1 Type system

Generalizing Hindley-Milner type inference algorithms
Heeren, B. J., Jurriaan Hage, and S. Doaitse Swierstra. "Generalizing Hindley-Milner type inference algorithms." (2002).
Description of the Hindley-Milner type system and inference algorithm. Also describes a constraint-solving algorithm.
HMF: Simple type inference for first-class polymorphism
Leijen, Daan. "HMF: Simple type inference for first-class polymorphism." ACM Sigplan Notices. Vol. 43. No. 9. ACM, 2008.
Describes an extension of Hindley-Milner that enables System F types including rank-N types and impredicative polymorphism.

3.2 Row polymorphism

A polymorphic type system for extensible records and variants
Gaster, Benedict R., and Mark P. Jones. "A polymorphic type system for extensible records and variants." (1996).
Describes a simple type system with (row polymorphic) extensible records and variants that only require lacks constraints.
Extensible records with scoped labels.
Leijen, Daan. "Extensible records with scoped labels." Trends in Functional Programming 5 (2005): 297-312.
Describes a very simple extension to Hindley-Milner that support extensible records and "scoped labels", which means labels can occur multiple times in a row. This requires no constraints at all.
First-class labels for extensible rows.
Leijen, D. J. P. "First-class labels for extensible rows." (2004).
Describes a type system where labels are first-class and one can define functions that take labels as arguments. This simplifies the language but complicates the type system.

3.3 Effect handlers

An effect system for algebraic effects and handlers.
Bauer, Andrej, and Matija Pretnar. "An effect system for algebraic effects

and handlers." International Conference on Algebra and Coalgebra in Computer Science. Springer, Berlin, Heidelberg, 2013.
Describes an effect system called "core Eff" which is an extension of a ML-style language with algebraic effects and handlers. The system is formalized in Twelf.

Programming with algebraic effects and handlers.
Bauer, Andrej, and Matija Pretnar. "Programming with algebraic effects and handlers." Journal of Logical and Algebraic Methods in Programming 84.1 (2015): 108-123.
Describes the programming language Eff, which is a ML-like language with algebraic effects and effect handlers.

An introduction to algebraic effects and handlers.
Pretnar, Matija. "An introduction to algebraic effects and handlers. invited tutorial paper." Electronic Notes in Theoretical Computer Science 319 (2015): 19-35.
This paper is a nice introduction to algebraic effects and handlers. It shows examples and gives semantics and typing rules.

Liberating effects with rows and handlers.
Hillerstrm, Daniel, and Sam Lindley. "Liberating effects with rows and handlers." Proceedings of the 1st International Workshop on Type-Driven Development. ACM, 2016.
Describes the Links programming language, which combines algebraic effects and handlers with row polymorphism. Includes a formalization.

Algebraic effects and effect handlers for idioms and arrows.
Lindley, Sam. "Algebraic effects and effect handlers for idioms and arrows." Proceedings of the 10th ACM SIGPLAN workshop on Generic programming. ACM, 2014.
Describes a generalization of algebraic effects that allows for other kinds of effectful computations.

Koka: Programming with row polymorphic effect types.
Leijen, Daan. "Koka: Programming with row polymorphic effect types." arXiv preprint arXiv:1406.2061 (2014).
Describes a programming language called Koka that has row polymorphic effect types.

Do Be Do Be Do: The Frank Programming Language
Lindley, Sam & McBride, Conor, "http://homepages.inf.ed.ac.uk/slindley/papers/frankly-draft-march2014.pdf"
Describes a programming language called Frank where every function is an

effect handler. Any function will implicitly work over effectful programs. Makes a distinction between value and computation types.

3.4 Implicits

The implicit calculus: a new foundation for generic programming
Oliveira, Bruno CdS, et al. "The implicit calculus: a new foundation for generic programming." ACM SIGPLAN Notices. Vol. 47. No. 6. ACM, 2012.
Describes a formalization of a minimal lambda calculus with implicits.
On the bright side of type classes: instance arguments in Agda.
Devriese, Dominique, and Frank Piessens. "On the bright side of type classes: instance arguments in Agda." ACM SIGPLAN Notices 46.9 (2011): 143-155.
Describes an implementation of implicits in Agda called instance arguments.
Modular implicits.
White, Leo, Frdric Bour, and Jeremy Yallop. "Modular implicits." arXiv preprint arXiv:1512.01895 (2015).
Describes an extension of OCaml that adds implicits to modules, enabling ad-hoc polymorphism.
Type classes as objects and implicits.
Oliveira, Bruno CdS, Adriaan Moors, and Martin Odersky. "Type classes as objects and implicits." ACM Sigplan Notices. Vol. 45. No. 10. ACM, 2010.
Describes how to implement type classes in Scala using it's objects and implicits parameters.