1 Syntax

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

l ranges of an infinite set of labels

π ::=		constraints
	τ / l	row lacks label
	value(τ)	type is not an effect type
	$\tau \sim \tau$	row equality

σ ::=		constrained type
	$\pi \Rightarrow \sigma$	constraint
	τ	type

τ ::=		types
	l	label
	c	type constructor
	α	type variable
	$\tau\ \tau$	type application
	$<>$	empty row
	$<\mathrm{l} : \tau \mid \tau>$	row extension
	$\mu\alpha\ .\ \tau$	recursive type
	$\forall\alpha\ .\ \sigma$	forall with constraint

$e ::=$		terms
	$'l$	label
	x	variable
	$e\ e$	application
	$\lambda x\ .\ e$	abstraction
	$\lambda(x : \tau)\ .\ e$	annotated abstraction
	let $x = e$ in e	let expression
	letr $x = e$ in e	recursive let expression
	$\{\}$	empty record
	handle $\{$ return $x \to x, l\ x\ k \to e, ...\}$	effect handling

1.1 Type aliases

$()$: Rec $<>$

1.2 Type constructors

Lab : Label $\to$ Type
Rec : Row $\to$ Type
Var : Row $\to$ Type
Eff : Row $\to$ Type $\to$ Type

1.3 Constants

$()$: Rec $<>$ unit
$(.) : \forall ltr\ .\ r/l\ \Rightarrow$ Lab $l\ \to$Rec$<\ l : t\ |\ r\ >\to t$ record selection
$(.+) : \forall ltr\ .\ r/l\ \Rightarrow$ Lab $l\ \to$Rec $r\ \to$Rec$<\ l : t\ |\ r\ >$ record extension
$(.-) : \forall ltr\ .\ r/l\ \Rightarrow$ Lab $l\ \to$Rec$<\ l : t\ |\ r\ >\to$Rec r record restriction
$(@) : \forall ltr\ .\ r/l\ \Rightarrow$ Lab $l \to t\ \to$Var$<\ l : t\ |\ r\ >$ variant injection
$(@+) : \forall ltr\ .\ r/l\ \Rightarrow$ Lab $l\ \to$Var $r\ \to$Var$<\ l : t\ |\ r\ >$ variant embedding
$(?) : \forall labr\ .\ r/l\ \Rightarrow$ Lab $l \to (a \to b) \to$ (Var $r \to b) \to$Var$<\ l : a\ |\ r\ >\to b$ variant elimination
end : $\forall t$. Var$<>\to t$ end variant elimination
$(!) : \forall labr\ .\ r/l\ \Rightarrow$ Lab $l \to a\ \to$Eff$<\ l : a \to b\ |\ r > b$ perform effect
bind : $\forall abr$. Eff $r\ a \to (a\ \to$Eff $r\ b)\ \to$Eff $r\ b$ effect sequencing
$(!+) : \forall labtr\ .\ r/l\ \Rightarrow$ Lab $l\ \to$Eff $r\ t\ \to$Eff$<\ l : a \to b\ |\ r\ >\ t$ effect

embedding

pure : $\forall rt$. Eff r $t \to t$	value from effect type
return : $\forall rt$. $t \to$Eff r t	value into effect type

2 Typing rules

Var

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

Gen

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau \\ \alpha \notin ftv(\Gamma)\end{array}}{\Gamma \vdash e : \forall \alpha . \tau}$$

Inst

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau_1 \\ \tau_1 \sqsubseteq \tau_2\end{array}}{\Gamma \vdash e : \tau_2}$$

Fun

$$\frac{\Gamma, x : \tau \vdash e : \rho}{\Gamma \vdash \lambda x \ . \ e : \tau \to \rho}$$

Fun-Ann

$$\frac{\Gamma, x : \tau \vdash e : \rho}{\Gamma \vdash \lambda (x : \tau) \ . \ e : \tau \to \rho}$$

Let

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \\ \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \\ \forall \tau_1' . \Gamma \vdash e_1 : \tau_1' \Rightarrow \tau_1 \sqsubseteq \tau_1'\end{array}}{\Gamma \vdash let\ x = e_1\ in\ e_2 : \tau_2}$$

Letr

$$\frac{\Gamma, x : \alpha \vdash e_1 : \tau_1 \qquad \alpha \sqsubseteq \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \qquad \forall \tau_1'. \Gamma \vdash e_1 : \tau_1' \Rightarrow \tau_1 \sqsubseteq \tau_1'}{\Gamma \vdash \mathit{letr}\ x = e_1\ \mathit{in}\ e_2 : \tau_2}$$

App

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_2 \to \tau \\ \Gamma \vdash e_2 : \tau_2 \\ (\forall \tau' \tau_2'. (\Gamma \vdash e_1 : \tau_2' \to \tau' \land \Gamma \vdash e_2 : \tau_2') \\ \Rightarrow [\![\tau_2 \to \tau]\!] \leq [\![\tau_2' \to \tau']\!])\end{array}}{\Gamma \vdash e_1\ e_2 : \tau}$$

3 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) == 11
?'Just (\x -> x + 1) (\_ -> 0) (@'Nothing {}) == 0

?'Just (\x -> x + 1) (\_ -> 0)
  : forall (r : Row) . Var < Just : Int | r > -> Int
?'Just (\x -> x + 1) end : Var < Just : Int > -> Int
```

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 :
  v ->
  Eff < Get : () -> v, Set : v -> () | r > t ->
  Eff r t
state = handle {
  Get _ k v -> k v v)
  Set v k _ -> k () v)
}

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

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

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

4 Questions

- How does impredicativity interact with row-polymorphic types or algebraic effects?
- How do higher-ranked types interact with row-polymorphic types or algebraic effects. (ST monad in Haskell?)
- Handlers that only handle one effect? (hypothesis: not as expressive as handlers with multiple effects)
- Best way to introduce recursive types in to the system? (equi-recursive or iso-recursive)
- Is a seperation between value types and computation types necessary? (call-by-push-value)

5 Papers

5.1 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.
Complete and easy bidirectional typechecking for higher-rank polymorphism.
Dunfield, Joshua, and Neelakantan R. Krishnaswami. "Complete and easy bidirectional typechecking for higher-rank polymorphism." ACM SIGPLAN Notices. Vol. 48. No. 9. ACM, 2013. A type system with System F types, including higher-ranked types and predicative instantiation. Contains bidirectional typing rules. Can subsume Hindley-Milner.
Ur: statically-typed metaprogramming with type-level record computation.

Chlipala, Adam. "Ur: statically-typed metaprogramming with type-level record computation." ACM Sigplan Notices. Vol. 45. No. 6. ACM, 2010.
Describes the programming Language Ur, which has advanced type-level computation on records.

5.2 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.

5.3 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.

Hillerström, 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.

Type directed compilation of row-typed algebraic effects.

Leijen, Daan. "Type directed compilation of row-typed algebraic effects." POPL. 2017.

Provides a nice up-to-date presentation of Koka, including algebraic effects and handlers.

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.