# Thesis report 3

Albert ten Napel

## 1 Syntax

The type system is based on System $F_\omega$ with types and kinds unified and row-polymorphic types with lacks constraints. We will talk about kinds if we mean types classifying types. *I have omitted first-class labels at the moment.*

$l$ taken from an infinite set of labels

| | |
|---|---|
| $\tau, \kappa ::=$ | (types) |
| $\alpha, \beta, \rho$ | (type variables) |
| $c_\tau$ | (type constants) |
| $C$ | (constraints) |
| $p$ | (proofs) |
| $\lambda(\alpha : \kappa).\tau$ | (type abstraction) |
| $\tau\ \tau$ | (type application) |
| $\tau[\kappa]$ | (kind application) |
| $\tau \rightarrow \tau$ | (type arrows) |
| $\forall(\alpha : \kappa).\tau$ | (universally quantified type) |
| $\langle\rangle$ | (empty row) |
| $\langle l : \tau \| \tau\rangle$ | (row extension) |

|  |  |
|---|---|
| $c_\tau ::=$ | (type constants) |
| $\quad$ *Sort* | (type of types of types of types) |
| $\quad$ *Constraint* | (type of constraints) |
| $\quad$ *Kind* | (type of types of types) |
| $\quad$ *Type* | (type of types) |
| $\quad$ *Row* | (type of rows) |
| $\quad$ *Bool* | (type of booleans) |
| $\quad$ *Rec* | (type constructor of records) |
| $\quad$ *Var* | (type constructor of variants) |
| $\quad$ *Eff* | (type constructor of effects) |

|  |  |
|---|---|
| $C ::=$ | (constraints) |
| $\quad \tau \backslash l$ | (row lacks label) |

|  |  |
|---|---|
| $p ::=$ | (proofs) |
| $\quad$ *LacksEmpty* $l$ | (empty row lacks label proof) |
| $\quad$ *LacksExtend* $l$ $l$ | (row extension lacks label proof) |

| | |
|---|---|
| $e ::=$ | (terms) |
| $x, y, z$ | (variables) |
| $c_e$ | (constants) |
| $e\ e$ | (application) |
| $\lambda(x : \tau).e$ | (abstraction) |
| $\Lambda(\alpha : \kappa).e$ | (term type abstraction) |
| $e[\tau]$ | (term type application) |
| $\{\}$ | (empty record) |
| $\mathit{select}\ l$ | (record selection) |
| $\mathit{extend}\ l$ | (record extension) |
| $\mathit{restrict}\ l$ | (record restriction) |
| $\mathit{inject}\ l$ | (variant injection) |
| $\mathit{embed}\ l$ | (variant embedding) |
| $\mathit{elim}\ l$ | (variant elimination) |
| $\mathit{end}$ | (variant elimination end) |
| $\mathit{perform}\ l$ | (perform effect) |
| $\mathit{effembed}\ l$ | (embed effect) |
| $\mathit{return}$ | (return) |
| $\mathit{pure}$ | (pure) |
| $\mathit{bind}$ | (bind) |
| $\mathit{handle}\{\mathit{return}\ s\ v \to e, l_1\ s\ v\ k \to e, ..., l_n\ s\ v\ k \to e\}$ | (handle effect) |
| $c_e ::=$ | (constants) |
| $\mathit{fix}$ | (fix (general recursion)) |
| $\mathit{true}$ | (true) |
| $\mathit{false}$ | (false) |
| $\Gamma ::=$ | (contexts) |
| $\varnothing$ | (empty) |
| $\Gamma, \alpha : \kappa$ | (type variable has kind) |
| $\Gamma, x : \tau$ | (variable has type) |

## 1.1 Syntactic sugar

### 1.1.1 Types

$$
\begin{aligned}
\tau_1 \to \tau_2 \to \tau_3 &:= \tau_1 \to (\tau_2 \to \tau_3) \\
\forall(\alpha : \tau_1)(\beta : \tau_2).\tau_3 &:= \forall(\alpha : \tau_1).\forall(\beta : \tau_2).\tau_3 \\
\tau_1 \Rightarrow \tau_2 &:= \forall(\_ : \tau_1).\tau_2 \\
\langle l : \tau \rangle &:= \langle l : \tau | \langle\rangle \rangle \\
\langle l_1 : \tau_1, l_2 : \tau_2 | \tau_3 \rangle &:= \langle l_1 : \tau_1 | \langle l_2 : \tau_2 | \tau_3 \rangle \rangle
\end{aligned}
$$

### 1.1.2 Terms

$$
\begin{aligned}
e_1\ e_2\ e_3 &:= (e_1\ e_2)\ e_3 \\
\lambda(x : \tau_1)(y : \tau_2).e &:= \lambda(x : \tau_1).\lambda(y : \tau_2).e \\
\Lambda(\alpha : \tau_1)(\beta : \tau_2).e &:= \Lambda(\alpha : \tau_1).\Lambda(\beta : \tau_2).e
\end{aligned}
$$

# 2 Typing rules

## 2.1 Judgments

$\Gamma \vdash \tau : \kappa$ (type $\tau$ is wellformed and has kind $\kappa$)
$\Gamma \vdash e : \tau$ (term $e$ has type $\tau$)

## 2.2 Row equivalence

We show the rules for equivalence of well-formed rows.
Eq-Refl

$$\frac{}{\rho \cong \rho}$$

Eq-Trans

$$\frac{\begin{array}{c}\rho_1 \cong \rho_2 \\ \rho_2 \cong \rho_3\end{array}}{\rho_1 \cong \rho_3}$$

Eq-Head

$$\frac{\rho_1 \cong \rho_2}{\langle l : \tau | \rho_1 \rangle \cong \langle l : \tau | \rho_2 \rangle}$$

Eq-Swap

$$\frac{l_1 \neq l_2}{\langle l_1 : \tau_1 | \langle l_2 : \tau_2 | \rho \rangle \rangle \cong \langle l_2 : \tau_2 | \langle l_1 : \tau_1 | \rho \rangle \rangle}$$

## 2.3 Kinding rules

First we define an order between kinds.
$Sort < Constraint < Kind$, $Kind < Type$ and $Kind < Row$.
Now we define the function $lgeKinds(\kappa_1, \kappa_2)$ to find the lowest greater element of two kinds using the relation above, the function is undefined if no such element exist.

Now we present the kinding rules for type constants. Notice that sort is a type of sort, which means the system is inconsistent. Since we will be adding general recursion later we are not worried about this.

K-Constraints

$$\frac{}{\Gamma \vdash C : \mathit{Constraint}}$$

K-Sort

$$\frac{}{\Gamma \vdash \mathit{Sort} : \mathit{Sort}}$$

K-Constraint

$$\frac{}{\Gamma \vdash \mathit{Constraint} : \mathit{Sort}}$$

K-Kind

$$\frac{}{\Gamma \vdash \mathit{Kind} : \mathit{Sort}}$$

K-Type

$$\frac{}{\Gamma \vdash \mathit{Type} : \mathit{Kind}}$$

K-Row

$$\frac{}{\Gamma \vdash \mathit{Row} : \mathit{Kind}}$$

K-Bool

$$\frac{}{\Gamma \vdash \mathit{Bool} : \mathit{Type}}$$

K-Record

$$\frac{}{\Gamma \vdash \mathit{Rec} : \mathit{Row} \to \mathit{Type}}$$

K-Variant

$$\frac{}{\Gamma \vdash \mathit{Var} : \mathit{Row} \to \mathit{Type}}$$

K-Eff

$$\frac{}{\Gamma \vdash \mathit{Eff} : \mathit{Row} \to \mathit{Type} \to \mathit{Type}}$$

Next the kinding rules for the proofs. There are only two proofs, a proof that the empty row does not contain any label and that row extension lacks a label if the labels are different.

K-LacksEmpty

$$\frac{}{\Gamma \vdash \mathit{LacksEmpty}\, l : \langle\rangle \backslash l}$$

K-LacksExtend

$$\frac{l_1 \neq l_2}{\Gamma \vdash \mathit{LacksExtend}\, l_1\ l_2 : \forall(\alpha : \mathit{Type})(\rho : \mathit{Row}).\rho\backslash l_1 \to \langle l_2 : \alpha | \rho\rangle \backslash l_1}$$

Lastly the kinding rules for the other types.

K-TVar

$$\frac{}{\Gamma, \alpha : \kappa \vdash \alpha : \kappa}$$

K-TAbs

$$\frac{\Gamma, \alpha : \kappa_1 \vdash \tau : \kappa_2}{\Gamma \vdash \lambda(\alpha : \kappa_1).\tau : \kappa_1 \to \kappa_2}$$

K-TApp

$$\frac{\begin{array}{l}\Gamma \vdash \tau_1 : \kappa_1 \to \kappa_2 \\ \Gamma \vdash \tau_2 : \kappa_1\end{array}}{\Gamma \vdash \tau_1\ \tau_2 : \kappa_2}$$

K-KApp

$$\frac{\begin{array}{l}\Gamma \vdash \tau_1 : \forall(\alpha : \kappa).\tau_3 \\ \Gamma \vdash \tau_2 : \kappa\end{array}}{\Gamma \vdash \tau_1[\tau_2] : [\tau_2/\alpha]\tau_3}$$

K-Arr

$$\frac{\begin{array}{c}\Gamma \vdash \tau_1 : \kappa_1 \\ \Gamma \vdash \tau_2 : \kappa_2\end{array}}{\Gamma \vdash \tau_1 \to \tau_2 : \mathit{lgeKinds}(\kappa_1, \kappa_2)}$$

K-Forall

$$\frac{\Gamma, \alpha : \kappa_1 \vdash \tau : \kappa_2}{\Gamma \vdash \forall(\alpha : \kappa_1).\tau : \kappa_2}$$

K-RowEmpty

$$\frac{}{\Gamma \vdash \langle\rangle : \mathit{Row}}$$

K-RowExtend

$$\frac{\begin{array}{c}\Gamma \vdash \tau_1 : \mathit{Type} \\ \Gamma \vdash \tau_2 : \mathit{Row}\end{array}}{\Gamma \vdash \langle l : \tau_1 | \tau_2 \rangle : \mathit{Row}}$$

## 2.4 Typing rules

Now we present the typing rules for terms.
T-True

$$\frac{}{\Gamma \vdash \mathit{true} : \mathit{Bool}}$$

T-False

$$\frac{}{\Gamma \vdash \mathit{false} : \mathit{Bool}}$$

T-Var

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

T-App

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \tau_1 \to \tau_2 \\ \Gamma \vdash e_2 : \tau_1\end{array}}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

T-Abs

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda(x : \tau_1).e : \tau_1 \to \tau_2}$$

T-TAbs

$$\frac{\Gamma, \alpha : \kappa \vdash e : \tau}{\Gamma \vdash \Lambda(\alpha : \kappa).e : \forall(\alpha : \kappa).\tau}$$

T-TApp

$$\frac{\begin{array}{l}\Gamma \vdash e : \forall(\alpha : \kappa).\tau_2 \\ \Gamma \vdash \tau_1 : \kappa\end{array}}{\Gamma \vdash e[\tau_1] : [\tau_1/\alpha]\tau_2}$$

Next operations on records and variants, note that the rows have lacks constraints so that duplicate labels are not possible.
T-EmptyRecord

$$\frac{}{\Gamma \vdash \{\} : \mathit{Rec}\ \langle\rangle}$$

T-Select

$$\frac{}{\Gamma \vdash \mathit{select}\, l : \forall(\alpha : \mathit{Type})(\rho : \mathit{Row}).\rho\backslash l \Rightarrow \mathit{Rec}\ \langle l : \alpha|\rho\rangle \rightarrow \alpha}$$

T-Extend

$$\frac{}{\Gamma \vdash \mathit{extend}\, l : \forall(\alpha : \mathit{Type})(\rho : \mathit{Row}).\rho\backslash l \Rightarrow \alpha \rightarrow \mathit{Rec}\ \rho \rightarrow \mathit{Rec}\ \langle l : \alpha|\rho\rangle}$$

T-Restrict

$$\frac{}{\Gamma \vdash \mathit{restrict}\, l : \forall(\alpha : \mathit{Type})(\rho : \mathit{Row}).\rho\backslash l \Rightarrow \mathit{Rec}\ \langle l : \alpha|\rho\rangle \rightarrow \mathit{Rec}\ \rho}$$

T-Inject

$$\frac{}{\Gamma \vdash \mathit{inject}\, l : \forall(\alpha : \mathit{Type})(\rho : \mathit{Row}).\rho\backslash l \Rightarrow \alpha \rightarrow \mathit{Var}\ \langle l : \alpha|\rho\rangle}$$

T-Embed

$$\frac{}{\Gamma \vdash \mathit{embed}\, l : \forall(\alpha : \mathit{Type})(\rho : \mathit{Row}).\rho\backslash l \Rightarrow \mathit{Var}\ \rho \rightarrow \mathit{Var}\ \langle l : \alpha|\rho\rangle}$$

T-Elim

$$\frac{}{\Gamma \vdash \mathit{elim}\, l : \forall(\alpha : \mathit{Type})(\beta : \mathit{Type})(\rho : \mathit{Row}).\rho\backslash l \Rightarrow (\alpha \to \beta) \to (\mathit{Var}\ \rho \to \beta) \to \mathit{Var}\ \langle l : \alpha | \rho\rangle \to \beta}$$

Notice here that any type can be returned, since it's impossible to create an empty variant.
T-End

$$\frac{}{\Gamma \vdash \mathit{end} : \forall(\alpha : \mathit{Type}).\mathit{Var}\ \langle\rangle \to \alpha}$$

Lastly the typing rules for the effects.
T-Perform

$$\frac{}{\Gamma \vdash \mathit{perform}\, l : \forall(\alpha : \mathit{Type})(\beta : \mathit{Type})(\rho : \mathit{Row}).\rho\backslash l \Rightarrow \alpha \to \mathit{Eff}\ \langle l : \alpha \to \beta | \rho\rangle\ \beta}$$

T-EffEmbed

$$\frac{}{\Gamma \vdash \mathit{effembed}\, l : \forall(\alpha : \mathit{Type})(\beta : \mathit{Type})(t : \mathit{Type})(\rho : \mathit{Row}).\rho\backslash l \Rightarrow \mathit{Eff}\ \rho\ t \to \mathit{Eff}\ \langle l : \alpha \to \beta | \rho\rangle\ t}$$

T-Return

$$\frac{}{\Gamma \vdash \mathit{return} : \forall(\alpha : \mathit{Type})(\rho : \mathit{Row}).\alpha \to \mathit{Eff}\ \rho\ \alpha}$$

T-Pure

$$\frac{}{\Gamma \vdash \mathit{pure} : \forall(\alpha : \mathit{Type}).\mathit{Eff}\ \langle\rangle\ \alpha \to \alpha}$$

T-Bind

$$\frac{}{\Gamma \vdash \mathit{bind} : \forall(\alpha : \mathit{Type})(\beta : \mathit{Type})(\rho : \mathit{Type}).(\alpha \to \mathit{Eff}\ \rho\ \beta) \to \mathit{Eff}\ \rho\ \alpha \to \mathit{Eff}\ \rho\ \beta}$$

Every handler is parameterized, meaning every handler can manipulate a single piece of state represented by variable $s$ of type $\sigma$.
T-Handle

$$\frac{\begin{array}{l}\Gamma, s : \sigma, v : \alpha \vdash e_r : \beta \\ [ \\ \quad \Gamma, s : \sigma, v : \alpha_i, k : (\sigma \to \beta_i \to \mathit{Eff}\ \rho\ \beta) \vdash e_i : \mathit{Eff}\ \rho\ \beta \\ ]_{0<i\leq n}\end{array}}{\begin{array}{l}\Gamma \vdash \mathit{handle}\{\mathit{return}\ s\ v \to e_r, l_1\ s\ v\ k \to e_1, ..., l_n\ s\ v\ k \to e_n\} : \\ \quad \sigma \to \mathit{Eff}\ \langle l_1 : \alpha_1 \to \beta_1, ..., l_n : \alpha_n \to \beta_n | \rho\rangle\ \alpha \to \mathit{Eff}\ \rho\ \beta\end{array}}$$

## 3 Examples

### 3.1 Formal examples

The type, definition and usage of the id function.
$id : \forall(t : \mathit{Type}).t \to t$
$id = \Lambda(t : \mathit{Type}).\lambda(x : t).t$
$id[\mathit{Bool}] : \mathit{Bool} \to \mathit{Bool}$

Creating the record $\{x = true\}$.Notice the explicit proof that is passed, proving that the empty row does not contain the label $x$.
$trueRecord : \mathit{Rec}\ \langle x : \mathit{Bool}\rangle$
$trueRecord = \mathit{extend}\ x[\mathit{Bool}][\langle\rangle][\mathit{LacksEmpty}\ x]\ \mathit{true}\ \{\}$

Adding another field $y$ to $trueRecord$: $\{y = false, x = true\}$. We need a bigger proof that shows that $trueRecord$ does not contain $y$.
$biggerRecord : \mathit{Rec}\ \langle y : \mathit{Bool}, x : \mathit{Bool}\rangle$
$biggerRecord =$
$\quad \mathit{extend}\ y[\mathit{Bool}][\langle x : \mathit{Bool}\rangle][\mathit{LacksExtend}\ y\ x[\mathit{Bool}][\langle\rangle]\ (\mathit{LacksEmpty}\ y)]\ \mathit{false}\ trueRecord$

## 3.2 Hypothetical examples

Following some examples in a hypothetical programming language based on the calculus presented. I will assume polymorphic arguments can be inferred.

Listing 1: Booleans as Variants

```
type alias Bool = Var { True : (), False : () }

true : Bool
true = inject True ()

false : Bool
false = inject False ()

elimBool : forall(t : Type) . t -> t -> Bool -> t
elimBool vt vf =
  elim True (\_ -> vt) (
  elim False (\_ -> vf)
  end)

not : Bool -> Bool
not = elimBool False True
```

Listing 2: State effect

```
// state handler returns a tuple of the final state and
// the resulting value of the computation
state : forall(s : Type) . s ->
  Eff <get : () -> s, set : s -> ()> v -> (s, v)
state = handle {
  return s v -> v,
  get s () k -> k s s, // the second arg to the callback k is
                       // the internal state value of the
                       // handler
  set s v k -> k v () // internal state is replaced by v
}
```

## 4 Possible extensions

- Type-level row operations, such as union, difference, row elimination (induction).
- Let rows be indexed by the kinds of types in the rows. Rows with higher kinds might be useful, if there are type-level row operations.
- Indexed algebraic effects (State effect where *set* can change the type of the value stored).
For example:

Listing 3: State effect that can change state type

```
// @ is used to indicate the pre- and post-types of the state
state : forall(s : Type) . s ->
  Eff <
    get : s@() -> s@s,
    set : forall(t : Type) . s@t -> t@()
  >
```

Listing 4: Typesafe state machine

```
// the possible states
type On = ()
type Off = ()

// you can only perform the turnOn effect if the state is Off
    and after the state will be On
type switchEffect = Eff <
  turnOn : Off@() -> On@()
  turnOff : On@() -> Off@()
> ()
```

## 5 Questions

- Will first-class labels have any useful interactions with algebraic effects?
- Handlers that only handle one effect? (hypothesis: not as expressive as handlers with multiple effects) A: No, the operations may interact and so the handlers cannot be decomposed
- 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) A: Computation types give cleaner mathematical semantics,

might be easier to work with

## 6 Papers

### 6.1 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.
**System FC with explicit kind equality**
*Weirich, Stephanie, Justin Hsu, and Richard A. Eisenberg. "System FC with explicit kind equality." ACM SIGPLAN Notices. Vol. 48. No. 9. ACM, 2013.*
Describes the core calculus of Haskell, System FC. Shows the advantages of unifying types and kinds. System FCs use if coercions might also be interesting for row-polymorphic types.

## 6.2 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.

## 6.3 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.