

# A fibrational view on computational effects

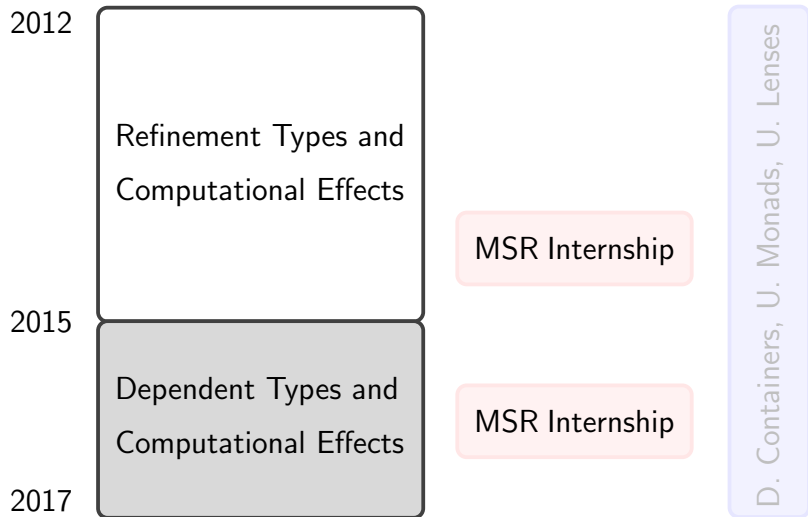
(an overview of my PhD thesis)

Danel Ahman

Prosecco Team, Inria Paris

Edinburgh, 28 November 2017

# Overview – how my PhD journey looked like



# Overview – dependent types

## The Curry-Howard correspondence:

Simple Types  $\sim$  Propositional Logic  $(\text{Nat}, \text{String}, \dots)$

Dependent Types  $\sim$  Predicate Logic  $(\Sigma, \Pi, =, \dots)$

**A tiny example:** we can use dep. types to express sorted lists

$$\ell : (\text{List Nat}) \vdash \text{Sorted}(\ell) \stackrel{\text{def}}{=} \forall i : \text{Nat} . (0 < i < \text{len } \ell) \Rightarrow (\ell[i-1] \leq \ell[i])$$

which in turn could be used to type a sorting function

$$\text{sort} : \forall \ell : (\text{List Nat}) . \exists \ell' : (\text{List Nat}) . \left( \text{Sorted}(\ell') \wedge \dots \right)$$

**Large examples:** CompCert (Coq), miTLS and HACL\* (F\*), ...

# Overview – dependent types

## The Curry-Howard correspondence:

Simple Types  $\sim$  Propositional Logic  $(\text{Nat}, \text{String}, \dots)$

Dependent Types  $\sim$  Predicate Logic  $(\Sigma, \Pi, =, \dots)$

**A tiny example:** we can use dep. types to express **sorted lists**

$$\ell : (\text{List Nat}) \vdash \text{Sorted}(\ell) \stackrel{\text{def}}{=} \forall i : \text{Nat} . (0 < i < \text{len } \ell) \Rightarrow (\ell[i-1] \leq \ell[i])$$

which in turn could be used to type a **sorting function**

$$\text{sort} : \forall \ell : (\text{List Nat}) . \exists \ell' : (\text{List Nat}) . \left( \text{Sorted}(\ell') \wedge \dots \right)$$

Large examples: CompCert (Coq), miTLS and HACL\* (F\*), ...

# Overview – dependent types

## The Curry-Howard correspondence:

Simple Types  $\sim$  Propositional Logic  $(\text{Nat}, \text{String}, \dots)$

Dependent Types  $\sim$  Predicate Logic  $(\Sigma, \Pi, =, \dots)$

**A tiny example:** we can use dep. types to express **sorted lists**

$$\ell : (\text{List Nat}) \vdash \text{Sorted}(\ell) \stackrel{\text{def}}{=} \forall i : \text{Nat} . (0 < i < \text{len } \ell) \Rightarrow (\ell[i-1] \leq \ell[i])$$

which in turn could be used to type a **sorting function**

$$\text{sort} : \forall \ell : (\text{List Nat}) . \exists \ell' : (\text{List Nat}) . \left( \text{Sorted}(\ell') \wedge \dots \right)$$

**Large examples:** CompCert (Coq), miTLS and HACL\* (F\*), ...

# Overview – computational effects

## Examples:

- state
- exceptions
- nondeterminism
- interactive IO
- ...

## Meta-languages and models: based on

- monads ( $\lambda_c$ ,  $\lambda_{ML}$ , FGCBV) (Moggi)

$$T : \mathcal{V} \longrightarrow \mathcal{V}$$

- adjunctions (CBPV, EEC) (Levy, Egger et al.)

$$F : \mathcal{V} \longrightarrow \mathcal{C} \quad U : \mathcal{C} \longrightarrow \mathcal{V}$$

- algebraic presentations (Plotkin and Power)

$$\text{get} : 1 \multimap S \quad \text{put} : S \multimap 1 \quad + \text{ equations}$$

# Overview – putting the two together

## We investigate the combination of

- dependent types  $(\Pi, \Sigma, V =_A W, \dots)$
- computational effects  $(\text{state, nondeterminism, IO, } \dots)$

## Two guiding problems

- effectful programs in types (e.g., get and put in types)
- types of effectful programs (e.g., of sequential composition)

## Our goals

- tell a mathematically natural story
- use established math. techniques
- cover a wide range of comp. effects
- discover smth. interesting

# Overview – putting the two together

## We investigate the combination of

- dependent types  $(\Pi, \Sigma, V =_A W, \dots)$
- computational effects  $(\text{state, nondeterminism, IO, } \dots)$

## Two guiding problems

- effectful programs in types (e.g., get and put in types)
- types of effectful programs (e.g., of sequential composition)

## Our goals

- tell a mathematically natural story (via a clean core language)
- use established math. techniques
- cover a wide range of comp. effects
- discover smth. interesting



# Overview – putting the two together

## We investigate the combination of

- dependent types  $(\Pi, \Sigma, V =_A W, \dots)$
- computational effects  $(\text{state, nondeterminism, IO, } \dots)$

## Two guiding problems

- effectful programs in types (e.g., get and put in types)
- types of effectful programs (e.g., of sequential composition)

## Our goals

- tell a mathematically natural story (via a clean core language)
- use established math. techniques (fibrations and adjunctions)
- cover a wide range of comp. effects
- discover smth. interesting

# Overview – putting the two together

## We investigate the combination of

- dependent types  $(\Pi, \Sigma, V =_A W, \dots)$
- computational effects  $(\text{state, nondeterminism, IO, } \dots)$

## Two guiding problems

- effectful programs in types (e.g., get and put in types)
- types of effectful programs (e.g., of sequential composition)

## Our goals

- tell a mathematically natural story (via a clean core language)
- use established math. techniques (fibrations and adjunctions)
- cover a wide range of comp. effects (alg. effects, continuations)
- discover smth. interesting

# Overview – putting the two together

## We investigate the combination of

- dependent types  $(\Pi, \Sigma, V =_A W, \dots)$
- computational effects  $(\text{state, nondeterminism, IO, } \dots)$

## Two guiding problems

- effectful programs in types (e.g., `get` and `put` in types)
- types of effectful programs (e.g., of sequential composition)

## Our goals

- tell a mathematically natural story (via a clean core language)
- use established math. techniques (fibrations and adjunctions)
- cover a wide range of comp. effects (alg. effects, continuations)
- discover smth. interesting (using handlers to reason about effects)

# **Effectful programs in types**

**(type-dependency in the presence of effects)**

# Effectful programs in types

**Q:** Should we allow situations such as  $\text{Sorted}[\text{receive}(y. M)/\ell]$ ?

**A1:** In this work, we say **not directly**

- types should only depend on static information about effects
- we allow dependency on effectful comps. via analysing **thunks**

**A2:** But we are also looking into the **direct** case

- type-dependency needs to be “homomorphic”
- intuitively, lift  $\text{Sorted}(\ell)$  to  $\text{Sorted}^\dagger(c)$ , where  $c: T(\text{List Chr})$

# Effectful programs in types

**Q:** Should we allow situations such as  $\text{Sorted}[\text{receive}(y.M)/\ell]$ ?

**A1:** In this work, we say **not directly**

- types should only depend on **static information** about effects
- we allow dependency on effectful comps. via analysing **thunks**

**A2:** But we are also looking into the **direct** case

- type-dependency needs to be “homomorphic”
- intuitively, lift  $\text{Sorted}(\ell)$  to  $\text{Sorted}^\dagger(c)$ , where  $c: T(\text{List Chr})$

# Effectful programs in types

**Q:** Should we allow situations such as  $\text{Sorted}[\text{receive}(y.M)/\ell]$ ?

**A1:** In this work, we say **not directly**

- types should only depend on **static information** about effects
- we allow dependency on effectful comps. via analysing **thunks**

**A2:** But we are also looking into the **direct** case

- type-dependency needs to be “**homomorphic**”
- intuitively, **lift**  $\text{Sorted}(\ell)$  **to**  $\text{Sorted}^\dagger(c)$ , **where**  $c : T(\text{List Chr})$

# Effectful programs in types

**Aim:** Types should only depend on static info about effects

**Solution:** CBPV/EEC style distinction between vals. and comps.

- value types  $\Gamma \vdash A$  (MLTT + thunks + ...)
- computation types  $\Gamma \vdash \underline{C}$  (dep. typed CBPV/EEC)
- where  $\Gamma$  contains only value variables  $x_1 : A_1, \dots, x_n : A_n$

Could have also considered Moggi's  $\lambda_{ML}$  and Levy's FGCBV

- building on CBPV/EEC gives a more general story
- especially for the treatment of sequential composition
- and also for integrating dependent- and effect-typing (ongoing)



# Effectful programs in types

**Aim:** Types should only depend on static info about effects

**Solution:** CBPV/EEC style distinction between vals. and comps.

- value types  $\Gamma \vdash A$  (MLTT + thunks + ...)
- computation types  $\Gamma \vdash \underline{C}$  (dep. typed CBPV/EEC)
- where  $\Gamma$  contains only value variables  $x_1 : A_1, \dots, x_n : A_n$

Could have also considered Moggi's  $\lambda_{ML}$  and Levy's FGCBV

- building on CBPV/EEC gives a more general story
- especially for the treatment of sequential composition
- and also for integrating dependent- and effect-typing (ongoing)

# Effectful programs in types

**Aim:** Types should only depend on static info about effects

**Solution:** CBPV/EEC style distinction between vals. and comps.

- value types  $\Gamma \vdash A$  (MLTT + thunks + ...)
- computation types  $\Gamma \vdash \underline{C}$  (dep. typed CBPV/EEC)
- where  $\Gamma$  contains only value variables  $x_1 : A_1, \dots, x_n : A_n$

Could have also considered Moggi's  $\lambda_{ML}$  and Levy's FGCBV

- building on CBPV/EEC gives a more general story
- especially for the treatment of sequential composition
- and also for integrating dependent- and effect-typing (ongoing)

# Assigning types to effectful programs

(e.g., sequential composition)

# Assigning types to effectful programs

**The problem:** The standard typing rule for seq. composition

$$\frac{\Gamma \vdash M : F A \quad \Gamma, x:A \vdash N : \underline{C}(x)}{\Gamma \vdash M \text{ to } x:A \text{ in } N : \underline{C}(x)}$$

is not correct any more because  $x$  can appear free in the type

$C$

in the conclusion

# Assigning types to effectful programs

**Aim:** To fix the typing rule of **sequential composition**

**Option 1:** We could restrict the free variables in  $\underline{C}$ : [Levy'04]

$$\frac{\Gamma \Vdash M : FA \quad \Gamma \vdash \underline{C} \quad \Gamma, x:A \Vdash N : \underline{C}}{\Gamma \Vdash M \text{ to } x:A \text{ in } N : \underline{C}}$$

**But:** Sometimes it is useful if  $\underline{C}$  can depend on  $x$ !

- say we consider

`fopen (return true, return false) to x:Bool in N`

- then it would be natural to let  $\underline{C}$  depend on  $x$ , e.g.,

$$x:\text{Bool} \vdash \underline{C}(x) \stackrel{\text{def}}{=} \text{if } x \text{ then "allow fread, fwrite, and fclose"} \\ \text{else "allow fopen"}$$

(needs more expressive comp. types than you see in this talk)

# Assigning types to effectful programs

**Aim:** To fix the typing rule of **sequential composition**

**Option 1:** We could **restrict the free variables** in  $\underline{C}$ : [Levy'04]

$$\frac{\Gamma \Vdash M : FA \quad \Gamma \vdash \underline{C} \quad \Gamma, x:A \Vdash N : \underline{C}}{\Gamma \Vdash M \text{ to } x:A \text{ in } N : \underline{C}}$$

**But:** Sometimes it is useful if  $\underline{C}$  can depend on  $x$ !

- say we consider

`fopen (return true, return false) to x:Bool in N`

- then it would be natural to let  $\underline{C}$  depend on  $x$ , e.g.,

$x:\text{Bool} \vdash \underline{C}(x) \stackrel{\text{def}}{=} \text{if } x \text{ then "allow fread, fwrite, and fclose"} \\ \text{else "allow fopen"}$

(needs more expressive comp. types than you see in this talk)

# Assigning types to effectful programs

**Aim:** To fix the typing rule of `sequential composition`

**Option 1:** We could `restrict the free variables` in  $\underline{C}$ : [Levy'04]

$$\frac{\Gamma \Vdash M : FA \quad \Gamma \vdash \underline{C} \quad \Gamma, x:A \Vdash N : \underline{C}}{\Gamma \Vdash M \text{ to } x:A \text{ in } N : \underline{C}}$$

**But:** Sometimes it is useful if  $\underline{C}$  can depend on  $x$ !

- say we consider

`fopen (return true, return false)` to  $x:\text{Bool}$  in  $N$

- then it would be natural to let  $\underline{C}$  depend on  $x$ , e.g.,

$x:\text{Bool} \vdash \underline{C}(x) \stackrel{\text{def}}{=} \text{if } x \text{ then "allow fread, fwrite, and fclose"}$   
else "allow fopen"

(needs more expressive comp. types than you see in this talk)

# Assigning types to effectful programs

**Aim:** To fix the typing rule of **sequential composition**

**Option 2:** One could lift sequential composition to type level

$$\Gamma \Vdash M \text{ to } x:A \text{ in } N : M \text{ to } x:A \text{ in } \underline{C}$$

**But:** Then comp. types would be singleton-like!?!

However, smth. like this is probably needed for the **direct** case.

**Option 3:** In the monadic metalanguage  $\lambda_{\text{ML}}$ , one could try

$$\frac{\Gamma \vdash M : T A \quad \Gamma, x:A \vdash N : T B(x)}{\Gamma \vdash M \text{ to } x:A \text{ in } N : T (\Sigma x:A. B)}$$

**But:** What makes this a principled solution? Why is it correct?



# Assigning types to effectful programs

**Aim:** To fix the typing rule of **sequential composition**

**Option 2:** One could **lift sequential composition** to type level

$$\Gamma \Vdash M \text{ to } x:A \text{ in } N : M \text{ to } x:A \text{ in } \underline{C}$$

**But:** Then comp. types would be singleton-like!?!

However, smth. like this is probably needed for the **direct** case.

**Option 3:** In the monadic metalanguage  $\lambda_{ML}$ , one could try

$$\frac{\Gamma \vdash M : T A \quad \Gamma, x:A \vdash N : T B(x)}{\Gamma \vdash M \text{ to } x:A \text{ in } N : T (\Sigma x:A. B)}$$

**But:** What makes this a principled solution? Why is it correct?

# Assigning types to effectful programs

**Aim:** To fix the typing rule of **sequential composition**

**Option 2:** One could **lift sequential composition** to type level

$$\Gamma \Vdash M \text{ to } x:A \text{ in } N : M \text{ to } x:A \text{ in } \underline{C}$$

**But:** Then comp. types would be singleton-like!?!

However, smth. like this is probably needed for the **direct** case.

**Option 3:** In the **monadic metalanguage**  $\lambda_{ML}$ , one could try

$$\frac{\Gamma \vdash M : T A \quad \Gamma, x:A \vdash N : T B(x)}{\Gamma \vdash M \text{ to } x:A \text{ in } N : T (\Sigma x : A. B)}$$

**But:** What makes this a principled solution? Why is it correct?

# Assigning types to effectful programs

**Aim:** To fix the typing rule of **sequential composition**

**Option 4:** We draw inspiration from algebraic effects

- and combine this with restricting  $\underline{C}$  in seq. comp. (**Option 1**)

E.g., consider the non-deterministic prog.  $\left(\text{for } x:\text{Nat} \models N : \underline{C}(x)\right)$

$$M \stackrel{\text{def}}{=} \text{choose}(\text{return } 4, \text{return } 2) \text{ to } x:\text{Nat} \text{ in } N$$

After making the non-det. choice, this program evaluates as either

$$N[4/x] : \underline{C}[4/x] \quad \text{or} \quad N[2/x] : \underline{C}[2/x]$$

**Idea:**  $M$  denotes an element of the coproduct of algebras

$$\underline{C}[4/x] + \underline{C}[2/x] \stackrel{\text{def}}{=} F\left(U(\underline{C}[4/x]) + U(\underline{C}[2/x])\right)_{/\equiv}$$

which we generalise to  $A$ -indexed coproducts, i.e., a comp.  $\Sigma$ -type

# Assigning types to effectful programs

**Aim:** To fix the typing rule of **sequential composition**

**Option 4:** We draw inspiration from **algebraic effects**

- and combine this with restricting  $\underline{C}$  in seq. comp. (**Option 1**)

E.g., consider the non-deterministic prog.  $(\text{for } x:\text{Nat} \models N : \underline{C}(x))$

$$M \stackrel{\text{def}}{=} \text{choose}(\text{return } 4, \text{return } 2) \text{ to } x:\text{Nat} \text{ in } N$$

After making the non-det. choice, this program evaluates as either

$$N[4/x] : \underline{C}[4/x] \quad \text{or} \quad N[2/x] : \underline{C}[2/x]$$

**Idea:**  $M$  denotes an element of the coproduct of algebras

$$\underline{C}[4/x] + \underline{C}[2/x] \stackrel{\text{def}}{=} F\left(U(\underline{C}[4/x]) + U(\underline{C}[2/x])\right)_{/\equiv}$$

which we generalise to  $A$ -indexed coproducts, i.e., a comp.  $\Sigma$ -type

# Assigning types to effectful programs

**Aim:** To fix the typing rule of **sequential composition**

**Option 4:** We draw inspiration from **algebraic effects**

- and combine this with restricting  $\underline{C}$  in seq. comp. (**Option 1**)

E.g., consider the **non-deterministic** prog.  $\left( \text{for } x:\text{Nat} \models N : \underline{C}(x) \right)$

$$M \stackrel{\text{def}}{=} \text{choose}(\text{return } 4, \text{return } 2) \text{ to } x:\text{Nat} \text{ in } N$$

After making the non-det. choice, this program evaluates as either

$$N[4/x] : \underline{C}[4/x] \quad \text{or} \quad N[2/x] : \underline{C}[2/x]$$

**Idea:**  $M$  denotes an element of the coproduct of algebras

$$\underline{C}[4/x] + \underline{C}[2/x] \stackrel{\text{def}}{=} F \left( U(\underline{C}[4/x]) + U(\underline{C}[2/x]) \right) /_{\equiv}$$

which we generalise to  $A$ -indexed coproducts, i.e., a comp.  $\Sigma$ -type

# Assigning types to effectful programs

**Aim:** To fix the typing rule of **sequential composition**

**Option 4:** We draw inspiration from **algebraic effects**

- and combine this with restricting  $\underline{C}$  in seq. comp. (**Option 1**)

E.g., consider the **non-deterministic** prog.  $\left( \text{for } x:\text{Nat} \models N : \underline{C}(x) \right)$

$$M \stackrel{\text{def}}{=} \text{choose}(\text{return } 4, \text{return } 2) \text{ to } x:\text{Nat} \text{ in } N$$

After **making the non-det. choice**, this program evaluates as either

$$N[4/x] : \underline{C}[4/x] \quad \text{or} \quad N[2/x] : \underline{C}[2/x]$$

**Idea:**  $M$  denotes an element of the coproduct of algebras

$$\underline{C}[4/x] + \underline{C}[2/x] \stackrel{\text{def}}{=} F \left( U(\underline{C}[4/x]) + U(\underline{C}[2/x]) \right) / \equiv$$

which we generalise to  $A$ -indexed coproducts, i.e., a comp.  $\Sigma$ -type

# Assigning types to effectful programs

**Aim:** To fix the typing rule of **sequential composition**

**Option 4:** We draw inspiration from **algebraic effects**

- and combine this with restricting  $\underline{C}$  in seq. comp. (**Option 1**)

E.g., consider the **non-deterministic** prog.  $\left( \text{for } x:\text{Nat} \models N : \underline{C}(x) \right)$

$$M \stackrel{\text{def}}{=} \text{choose}(\text{return } 4, \text{return } 2) \text{ to } x:\text{Nat} \text{ in } N$$

After **making the non-det. choice**, this program evaluates as either

$$N[4/x] : \underline{C}[4/x] \quad \text{or} \quad N[2/x] : \underline{C}[2/x]$$

**Idea:**  $M$  denotes an element of the **coproduct of algebras**

$$\underline{C}[4/x] + \underline{C}[2/x] \stackrel{\text{def}}{=} F \left( U(\underline{C}[4/x]) + U(\underline{C}[2/x]) \right) /_{\equiv}$$

which we generalise to  $A$ -indexed coproducts, i.e., a **comp.  $\Sigma$ -type**

## Putting these ideas together

(eMLTT: a core dep.-typed language with comp. effects)



# eMLTT – value and comp. types

**Value types:** MLTT + *thunks* + ...

$A, B ::= \text{Nat} \mid 1 \mid 0 \mid \Pi x:A. B \mid \Sigma x:A. B \mid V=_A W \mid \underline{UC} \mid \dots$

- $\underline{UC}$  is the type of *thunked* (i.e., suspended) *computations*

**Computation types:** dep.-typed version of EEC's comp. types

$\underline{C}, \underline{D} ::= FA \mid \Pi x:A. \underline{C} \mid \Sigma x:A. \underline{C}$

- $FA$  is the type of computations returning values of type  $A$
- $\Pi x:A. \underline{C}$  is the type of dependent effectful functions
  - generalises CBPV/EEC's comp. types  $A \rightarrow \underline{C}$  and  $\underline{C} \times \underline{D}$
- $\Sigma x:A. \underline{C}$  is the type of dep. pairs of values and effectful comps.
  - captures the intuition about seq. comp. and coprods. of algebras
  - generalises EEC's comp. types  $!A \otimes \underline{C}$  and  $\underline{C} \oplus \underline{D}$

# eMLTT – value and comp. types

**Value types:** MLTT + *thunks* + ...

$A, B ::= \text{Nat} \mid 1 \mid 0 \mid \Pi x:A. B \mid \Sigma x:A. B \mid V=_A W \mid \underline{UC} \mid \dots$

- $\underline{UC}$  is the type of *thunked* (i.e., suspended) *computations*

**Computation types:** dep.-typed version of EEC's comp. types

$\underline{C}, \underline{D} ::= FA \mid \Pi x:A. \underline{C} \mid \Sigma x:A. \underline{C}$

- $FA$  is the type of computations *returning values* of type  $A$
- $\Pi x:A. \underline{C}$  is the type of dependent *effectful functions*
  - generalises CBPV/EEC's comp. types  $A \rightarrow \underline{C}$  and  $\underline{C} \times \underline{D}$
- $\Sigma x:A. \underline{C}$  is the type of *dep. pairs* of values and effectful comps.
  - captures the intuition about seq. comp. and *coprods. of algebras*
  - generalises EEC's comp. types  $!A \otimes \underline{C}$  and  $\underline{C} \oplus \underline{D}$

# eMLTT – value and comp. terms

**Value terms:** MLTT + *thunks* + ...

$$V, W ::= x \mid \text{zero} \mid \text{succ } V \mid \dots \mid \text{thunk } M \mid \dots$$

- equational theory based on *intensional* MLTT

**Comp. terms:** dep.-typed version of CBPV/EEC's comp. terms

$$\begin{array}{lcl} M, N ::= & \text{force } V & \\ & \text{return } V & \\ & M \text{ to } x:A \text{ in } N & \\ & \lambda x:A. M & \\ & MV & \\ & \langle V, M \rangle & \text{(comp. } \Sigma \text{ intro.)} \\ & M \text{ to } \langle x:A, z:\underline{C} \rangle \text{ in } K & \text{(comp. } \Sigma \text{ elim.)} \end{array}$$

**But:** Value and comp. terms alone do not suffice, as in EEC!

# eMLTT – value and comp. terms

**Value terms:** MLTT + *thunks* + ...

$$V, W ::= x \mid \text{zero} \mid \text{succ } V \mid \dots \mid \text{thunk } M \mid \dots$$

- equational theory based on *intensional* MLTT

**Comp. terms:** dep.-typed version of CBPV/EEC's comp. terms

$$\begin{array}{ll} M, N ::= & \text{force } V \\ & \mid \text{return } V \\ & \mid M \text{ to } x:A \text{ in } N \\ & \mid \lambda x:A. M \\ & \mid MV \\ & \mid \langle V, M \rangle & (\text{comp. } \Sigma \text{ intro.}) \\ & \mid M \text{ to } \langle x:A, z:\underline{C} \rangle \text{ in } K & (\text{comp. } \Sigma \text{ elim.}) \end{array}$$

**But:** Value and comp. terms alone do not suffice, as in EEC!

# eMLTT – value and comp. terms

**Value terms:** MLTT + *thunks* + ...

$$V, W ::= x \mid \text{zero} \mid \text{succ } V \mid \dots \mid \text{thunk } M \mid \dots$$

- equational theory based on *intensional* MLTT

**Comp. terms:** dep.-typed version of CBPV/EEC's comp. terms

$$\begin{array}{ll} M, N ::= & \text{force } V \\ & \mid \text{return } V \\ & \mid M \text{ to } x:A \text{ in } N \\ & \mid \lambda x:A. M \\ & \mid MV \\ & \mid \langle V, M \rangle & (\text{comp. } \Sigma \text{ intro.}) \\ & \mid M \text{ to } \langle x:A, z:\underline{C} \rangle \text{ in } K & (\text{comp. } \Sigma \text{ elim.}) \end{array}$$

**But:** Value and comp. terms alone do not suffice, as in EEC!

# eMLTT – homomorphism terms

**Note:** We need to define  $K$  in such a way that the intended left-to-right evaluation order is preserved, e.g., consider

$$\Gamma \Vdash \langle V, M \rangle \text{ to } \langle x:A, z:\underline{C} \rangle \text{ in } K = K[V/x, M/z] : \underline{D}$$

**Homomorphism terms:** dep.-typed version of EEC's linear terms

$$\begin{array}{ll} K, L ::= & z \quad \text{(linear comp. vars.)} \\ & K \text{ to } x:A \text{ in } M \\ & \lambda x:A. K \\ & KV \\ & \langle V, K \rangle \quad \text{(comp. } \Sigma \text{ intro.)} \\ & K \text{ to } \langle x:A, z:\underline{C} \rangle \text{ in } L \quad \text{(comp. } \Sigma \text{ elim.)} \end{array}$$

**Typing judgments:**

- $\Gamma \Vdash V : A$
- $\Gamma \Vdash M : \underline{C}$
- $\Gamma \mid z:\underline{C} \Vdash K : \underline{D}$  (linear in  $z$ ; comp. bound to  $z$  happens first)

# eMLTT – homomorphism terms

**Note:** We need to define  $K$  in such a way that the intended left-to-right evaluation order is preserved, e.g., consider

$$\Gamma \Vdash \langle V, M \rangle \text{ to } \langle x:A, z:\underline{C} \rangle \text{ in } K = K[V/x, M/z] : \underline{D}$$

**Homomorphism terms:** dep.-typed version of EEC's linear terms

$$\begin{array}{ll} K, L ::= & z \quad \text{(linear comp. vars.)} \\ & | \quad K \text{ to } x:A \text{ in } M \\ & | \quad \lambda x:A. K \\ & | \quad KV \\ & | \quad \langle V, K \rangle \quad \text{(comp. } \Sigma \text{ intro.)} \\ & | \quad K \text{ to } \langle x:A, z:\underline{C} \rangle \text{ in } L \quad \text{(comp. } \Sigma \text{ elim.)} \end{array}$$

**Typing judgments:**

- $\Gamma \Vdash V : A$
- $\Gamma \Vdash M : \underline{C}$
- $\Gamma \mid z:\underline{C} \Vdash K : \underline{D}$  (linear in  $z$ ; comp. bound to  $z$  happens first)

# eMLTT – typing sequential composition

We can then account for **type-dependency in seq. comp.** as

$$\frac{\Gamma \Vdash M : F A \quad \Gamma \vdash \Sigma x:A. \underline{C}(x) \quad \frac{\Gamma, x:A \Vdash N : \underline{C}(x)}{\Gamma, x:A \Vdash \langle x, N \rangle : \Sigma x:A. \underline{C}(x)}}{\Gamma \Vdash M \text{ to } x:A \text{ in } \langle x, N \rangle : \Sigma x:A. \underline{C}(x)}$$

The **seq. comp. rule for  $\lambda_{ML}$**  is justified by the type isomorphism

$$\frac{\Gamma \vdash A \quad \Gamma, x:A \vdash B(x)}{\Gamma \vdash U\left(\Sigma x:A. F B(x)\right) \cong UF\left(\Sigma x:A. B(x)\right) = T\left(\Sigma x:A. B(x)\right)}$$



# Categorical semantics of eMLTT

(fibrations + adjunctions)

# Fibred adjunction models – value part

Given by a **split closed comprehension category**  $p$ , as in

$$\begin{array}{ccc} & \mathcal{V} & \\ p \swarrow & \uparrow 1 & \searrow \{-\} \\ & \mathcal{B} & \end{array}$$

allowing us to define a **partial interpretation fun.**  $\llbracket - \rrbracket$ , that maps:

- a **context**  $\Gamma$  to an object  $\llbracket \Gamma \rrbracket$  in  $\mathcal{B}$ , with
  - $\llbracket \diamond \rrbracket \stackrel{\text{def}}{=} 1$
  - $\llbracket \Gamma, x:A \rrbracket \stackrel{\text{def}}{=} \{\llbracket \Gamma; A \rrbracket\}$  (if  $x \notin \text{Vars}(\Gamma)$  and  $\llbracket \Gamma; A \rrbracket$  is defined)
- a context  $\Gamma$  and a **value type**  $A$  to an object  $\llbracket \Gamma; A \rrbracket$  in  $\mathcal{V}_{\llbracket \Gamma \rrbracket}$
- a context  $\Gamma$  and a **value term**  $V$  to  $\llbracket \Gamma; V \rrbracket : 1_{\llbracket \Gamma \rrbracket} \longrightarrow A$  in  $\mathcal{V}_{\llbracket \Gamma \rrbracket}$

# Fibred adjunction models – value part

Given by a split closed comprehension category  $p$ , as in

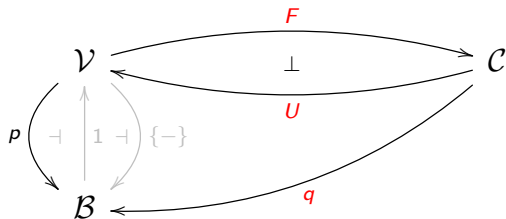
$$\begin{array}{c} \mathcal{V} \\ \begin{array}{c} \curvearrowleft \quad \dashv \\ \uparrow 1 \\ \downarrow \dashv \quad \curvearrowright \end{array} \\ \mathcal{B} \end{array} \quad \{-\}$$

such that

- $p$  has split fibred strong colimits of shape **0** and **2** [Jacobs'99]
  - (in thesis, also Jacobs-style axiomatisation for arbitrary shapes)
- $p$  has weak split fibred strong natural numbers
  - (axiomatisation is given in the style of fibrational induction)
- $p$  has split intensional propositional equality
  - (currently very synthetic ax., would like a weak form of adjoints)

# Fibred adjunction models – effects part

Given by a **split fibration**  $q$  and a split fib. adjunction  $F \dashv U$ , as in



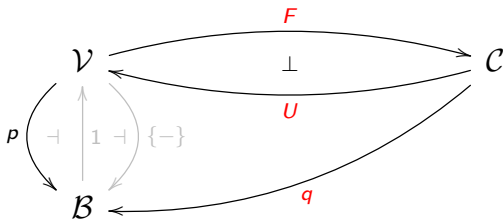
we extend the **partial interpretation fun.**  $\llbracket - \rrbracket$  so that it maps:

- a ctx.  $\Gamma$  and a **comp. type**  $\underline{C}$  to an object  $\llbracket \Gamma; \underline{C} \rrbracket$  in  $\mathcal{C}_{\llbracket \Gamma \rrbracket}$
- a ctx.  $\Gamma$  and a **comp. term**  $M$  to  $\llbracket \Gamma; M \rrbracket : 1_{\llbracket \Gamma \rrbracket} \longrightarrow U(\underline{C})$  in  $\mathcal{V}_{\llbracket \Gamma \rrbracket}$
- a ctx.  $\Gamma$ , a comp. var.  $z$ , a comp. type  $\underline{C}$ , and a **hom. term**  $K$  to

$$\llbracket \Gamma; z : \underline{C}; K \rrbracket : \llbracket \Gamma; \underline{C} \rrbracket \longrightarrow \underline{D} \text{ in } \mathcal{C}_{\llbracket \Gamma \rrbracket}$$

# Fibred adjunction models – effects part

Given by a **split fibration**  $q$  and a split fib. adjunction  $F \dashv U$ , as in



such that

- $q$  has split dependent  $p$ -products (**comp.**  $\Pi$ -type; r. adj. to wk.)
- $q$  has split dependent  $p$ -coproducts (**comp.**  $\Sigma$ -type; l. adj. to wk.)

and to account for the full calculus presented in the thesis,

- $q$  admits split fibred pre-enrichment in  $p$  (**hom.** function type  $\multimap$ )

# Fibred adjunction models – correctness

**Theorem** (Soundness):

- If  $\Gamma \vdash \underline{C}$ , then  $\llbracket \Gamma; \underline{C} \rrbracket \in \mathcal{C}_{\llbracket \Gamma \rrbracket}$
- If  $\Gamma \Vdash M : \underline{C}$ , then  $\llbracket \Gamma; M \rrbracket : 1_{\llbracket \Gamma \rrbracket} \longrightarrow U(\llbracket \Gamma; \underline{C} \rrbracket)$
- If  $\Gamma \mid z : \underline{C} \Vdash K : \underline{D}$ , then  $\llbracket \Gamma; z : \underline{C}; K \rrbracket : \llbracket \Gamma; \underline{C} \rrbracket \longrightarrow \llbracket \Gamma; \underline{D} \rrbracket$
- If  $\Gamma \vdash \underline{C} = \underline{D}$ , then  $\llbracket \Gamma; \underline{C} \rrbracket = \llbracket \Gamma; \underline{D} \rrbracket \in \mathcal{C}_{\llbracket \Gamma \rrbracket}$
- ...

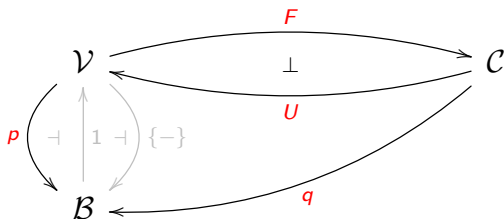
**Theorem** (Classifying model):

- The well-formed syntax of eMLTT forms a fib. adjunction model.

**Theorem** (Completeness):

- If two types or terms are equal in all fibred adjunction models, then they are also equal in the equational theory of eMLTT.

## Examples of fibred adjunction models



# Examples of fibred adjunction models

## Example 1 (identity adjunctions):

- sound as long as we haven't included any actual comp. effects

## Example 2 (simple fibrations from enriched adj. models of EEC):

- doesn't support any real type dependency (constant families)

## Example 3 (families fibrations and lifting of adjunctions):

- $([\Gamma], [A]) \in \text{Fam}(\text{Set})$  (where  $[A] \in [\Gamma] \rightarrow \text{Set}$ )
- $([\Gamma], [\underline{C}]) \in \text{Fam}(\mathcal{D})$  (where  $[\underline{C}] \in [\Gamma] \rightarrow \mathcal{D}$ )

## Example 4 (continuous families and CPO-enriched monads):

- $([\Gamma], [A]) \in \text{CFam}(\text{CPO})$  (where  $[A] \in [\Gamma] \rightarrow \text{CPO}^{EP}$ )
- $([\Gamma], [\underline{C}]) \in \text{CFam}(\text{CPO}^T)$  (where  $[\underline{C}] \in [\Gamma] \rightarrow (\text{CPO}^T)^{EP}$ )
- **Theorem:**  $\text{cod}_{\text{CPO}}$  is not suitable because CPO is not an LCCC.



# Examples of fibred adjunction models

## Example 1 (identity adjunctions):

- sound as long as we haven't included any actual comp. effects

## Example 2 (simple fibrations from enriched adj. models of EEC):

- doesn't support any real type dependency (constant families)

## Example 3 (families fibrations and lifting of adjunctions):

- $([\Gamma], [A]) \in \text{Fam}(\text{Set})$  (where  $[A] \in [\Gamma] \rightarrow \text{Set}$ )
- $([\Gamma], [\underline{C}]) \in \text{Fam}(\mathcal{D})$  (where  $[\underline{C}] \in [\Gamma] \rightarrow \mathcal{D}$ )

## Example 4 (continuous families and CPO-enriched monads):

- $([\Gamma], [A]) \in \text{CFam}(\text{CPO})$  (where  $[A] \in [\Gamma] \rightarrow \text{CPO}^{\text{EP}}$ )
- $([\Gamma], [\underline{C}]) \in \text{CFam}(\text{CPO}^{\text{T}})$  (where  $[\underline{C}] \in [\Gamma] \rightarrow (\text{CPO}^{\text{T}})^{\text{EP}}$ )
- **Theorem:**  $\text{cod}_{\text{CPO}}$  is not suitable because CPO is not an LCCC.

# Examples of fibred adjunction models

## Example 1 (identity adjunctions):

- sound as long as we haven't included any actual comp. effects

## Example 2 (simple fibrations from enriched adj. models of EEC):

- doesn't support any real type dependency (constant families)

## Example 3 (families fibrations and lifting of adjunctions):

- $([\Gamma], [A]) \in \text{Fam}(\text{Set})$  (where  $[A] \in [\Gamma] \longrightarrow \text{Set}$ )
- $([\Gamma], [\underline{C}]) \in \text{Fam}(\mathcal{D})$  (where  $[\underline{C}] \in [\Gamma] \longrightarrow \mathcal{D}$ )

## Example 4 (continuous families and CPO-enriched monads):

- $([\Gamma], [A]) \in \text{CFam}(\text{CPO})$  (where  $[A] \in [\Gamma] \longrightarrow \text{CPO}^{EP}$ )
- $([\Gamma], [\underline{C}]) \in \text{CFam}(\text{CPO}^T)$  (where  $[\underline{C}] \in [\Gamma] \longrightarrow (\text{CPO}^T)^{EP}$ )
- **Theorem:**  $\text{cod}_{\text{CPO}}$  is not suitable because CPO is not an LCCC.

# Examples of fibred adjunction models

## Example 1 (identity adjunctions):

- sound as long as we haven't included any actual comp. effects

## Example 2 (simple fibrations from enriched adj. models of EEC):

- doesn't support any real type dependency (constant families)

## Example 3 (families fibrations and lifting of adjunctions):

- $([\Gamma], [A]) \in \text{Fam}(\text{Set})$  (where  $[A] \in [\Gamma] \rightarrow \text{Set}$ )
- $([\Gamma], [\underline{C}]) \in \text{Fam}(\mathcal{D})$  (where  $[\underline{C}] \in [\Gamma] \rightarrow \mathcal{D}$ )

## Example 4 (continuous families and CPO-enriched monads):

- $([\Gamma], [A]) \in \text{CFam}(\text{CPO})$  (where  $[A] \in [\Gamma] \rightarrow \text{CPO}^{\text{EP}}$ )
- $([\Gamma], [\underline{C}]) \in \text{CFam}(\text{CPO}^{\mathbf{T}})$  (where  $[\underline{C}] \in [\Gamma] \rightarrow (\text{CPO}^{\mathbf{T}})^{\text{EP}}$ )
- **Theorem:**  $\text{cod}_{\text{CPO}}$  is not suitable because CPO is not an LCCC.

# Examples of fibred adjunction models

**Example 5** (EM-resolutions of split fibred monads):

- given a **split fibred monad**  $\mathbf{T} = (T, \eta, \mu)$  on  $\mathcal{P}$ , i.e.,

$$\begin{array}{ccc}
 \mathcal{V} & \xrightarrow{T} & \mathcal{V} \\
 \searrow \mathcal{P} & & \swarrow \mathcal{P} \\
 & \mathcal{B} &
 \end{array}
 \quad \text{and} \quad
 \mathcal{P}(\eta_A) = \text{id}_{\mathcal{P}(A)} \quad \mathcal{P}(\mu_A) = \text{id}_{\mathcal{P}(A)}$$

- we consider models based on the **EM-resolution** of  $\mathbf{T}$

$$\begin{array}{ccc}
 \mathcal{V} & \begin{array}{c} \xrightarrow{F^T} \\ \perp \\ \xleftarrow{U^T} \end{array} & \mathcal{V}^T \\
 \searrow \mathcal{P} & & \swarrow \mathcal{P}^T \\
 & \mathcal{B} &
 \end{array}$$

where  $(A \in \mathcal{V}, \alpha : T(A) \longrightarrow A) \in \mathcal{V}^T$

- and show that **three familiar results** hold for this situation

# Examples of fibred adjunction models

**Example 5** (EM-resolutions of split fibred monads):

- **Theorem 1:** If  $p$  supports  $\Pi$ -types, then  $p^T$  also supports  $\Pi$ -types

$$\Pi_A^T(B, \beta) \stackrel{\text{def}}{=} (\Pi_A(B), \beta_{\Pi_A^T})$$

- **Prop.:** If  $p$  supports  $\Sigma$ -types, then  $T$  has a dependent strength

$$\sigma_A : \Sigma_A \circ T \longrightarrow T \circ \Sigma_A \quad (A \in \mathcal{V})$$

- **Theorem 2:** If  $\sigma_A$  are natural isos., then  $p^T$  supports  $\Sigma$ -types

$$\Sigma_A^T(B, \beta) \stackrel{\text{def}}{=} (\Sigma_A(B), \beta_{\Sigma_A^T})$$

- **Theorem 3:** If  $p$  supports  $\Sigma$ -types and  $p^T$  has split fibred reflexive coequalizers, then  $p^T$  also supports  $\Sigma$ -types

$$\Sigma_A^T(B, \beta) \stackrel{\text{def}}{=} F^T(\Sigma_A(B)) /_{\equiv}$$

# Examples of fibred adjunction models

**Example 5** (EM-resolutions of split fibred monads):

- **Theorem 1:** If  $p$  supports  $\Pi$ -types, then  $p^{\mathbf{T}}$  also supports  $\Pi$ -types

$$\Pi_A^{\mathbf{T}}(B, \beta) \stackrel{\text{def}}{=} (\Pi_A(B), \beta_{\Pi_A^{\mathbf{T}}})$$

- **Prop.:** If  $p$  supports  $\Sigma$ -types, then  $\mathbf{T}$  has a **dependent strength**

$$\sigma_A : \Sigma_A \circ T \longrightarrow T \circ \Sigma_A \quad (A \in \mathcal{V})$$

- **Theorem 2:** If  $\sigma_A$  are **natural isos.**, then  $p^{\mathbf{T}}$  supports  $\Sigma$ -types

$$\Sigma_A^{\mathbf{T}}(B, \beta) \stackrel{\text{def}}{=} (\Sigma_A(B), \beta_{\Sigma_A^{\mathbf{T}}})$$

- **Theorem 3:** If  $p$  supports  $\Sigma$ -types and  $p^{\mathbf{T}}$  has split fibred reflexive coequalizers, then  $p^{\mathbf{T}}$  also supports  $\Sigma$ -types

$$\Sigma_A^{\mathbf{T}}(B, \beta) \stackrel{\text{def}}{=} F^{\mathbf{T}}(\Sigma_A(B)) /_{\equiv}$$

# Examples of fibred adjunction models

**Example 5** (EM-resolutions of split fibred monads):

- **Theorem 1:** If  $p$  supports  $\Pi$ -types, then  $p^{\mathbf{T}}$  also supports  $\Pi$ -types

$$\Pi_A^{\mathbf{T}}(B, \beta) \stackrel{\text{def}}{=} (\Pi_A(B), \beta_{\Pi_A^{\mathbf{T}}})$$

- **Prop.:** If  $p$  supports  $\Sigma$ -types, then  $\mathbf{T}$  has a **dependent strength**

$$\sigma_A : \Sigma_A \circ T \longrightarrow T \circ \Sigma_A \quad (A \in \mathcal{V})$$

- **Theorem 2:** If  $\sigma_A$  are **natural isos.**, then  $p^{\mathbf{T}}$  supports  $\Sigma$ -types

$$\Sigma_A^{\mathbf{T}}(B, \beta) \stackrel{\text{def}}{=} (\Sigma_A(B), \beta_{\Sigma_A^{\mathbf{T}}})$$

- **Theorem 3:** If  $p$  supports  $\Sigma$ -types and  $p^{\mathbf{T}}$  has **split fibred reflexive coequalizers**, then  $p^{\mathbf{T}}$  also supports  $\Sigma$ -types

$$\Sigma_A^{\mathbf{T}}(B, \beta) \stackrel{\text{def}}{=} F^{\mathbf{T}}(\Sigma_A(B)) /_{\equiv}$$

# **Algebraic effects**

**(operations and equations)**



# Algebraic effects – ops. and eqs.

**Fibred effect theories**  $\mathcal{T}_{\text{eff}}$ :

- signatures of **dependently typed operation symbols**

$$\frac{\cdot \vdash I \quad x_i : I \vdash O \quad I \text{ and } O \text{ are pure value types}}{\text{op} : (x_i : I) \multimap O}$$

- equipped with **equations** on derivable effect terms

**In eMLTT:**

$$M ::= \dots \mid \text{op}_V^{\underline{C}}(x.M)$$

**General algebraicity equations** (in addition to eff. th. eqs.):

$$\frac{\Gamma \Vdash V : I \quad \Gamma, x : O[V/x_i] \Vdash M : \underline{C} \quad \Gamma \mid z : \underline{C} \Vdash K : \underline{D}}{\Gamma \Vdash K[\text{op}_V^{\underline{C}}(x.M)/z] = \text{op}_V^{\underline{D}}(x.K[M/z]) : \underline{D}} \quad (\text{op} : (x_i : I) \multimap O)$$

**Sound semantics:** Based on families fibrations and Law. theories

- $p : \text{Fam}(\text{Set}) \longrightarrow \text{Set}$  and  $q : \text{Fam}(\text{Mod}(\mathcal{L}_{\mathcal{T}_{\text{eff}}}, \text{Set})) \longrightarrow \text{Set}$

# Algebraic effects – ops. and eqs.

**Fibred effect theories**  $\mathcal{T}_{\text{eff}}$ :

- signatures of **dependently typed operation symbols**

$$\frac{\cdot \vdash I \quad x_i : I \vdash O \quad I \text{ and } O \text{ are pure value types}}{\text{op} : (x_i : I) \multimap O}$$

- equipped with **equations** on derivable effect terms

**In eMLTT:**

$$M ::= \dots \mid \text{op}_V^C(x.M)$$

**General algebraicity equations** (in addition to eff. th. eqs.):

$$\frac{\Gamma \Vdash V : I \quad \Gamma, x : O[V/x_i] \Vdash M : \underline{C} \quad \Gamma \mid z : \underline{C} \Vdash K : \underline{D}}{\Gamma \Vdash K[\text{op}_V^C(x.M)/z] = \text{op}_V^D(x.K[M/z]) : \underline{D}} \quad (\text{op} : (x_i : I) \multimap O)$$

**Sound semantics:** Based on families fibrations and Law. theories

- $p : \text{Fam}(\text{Set}) \longrightarrow \text{Set}$  and  $q : \text{Fam}(\text{Mod}(\mathcal{L}_{\mathcal{T}_{\text{eff}}}, \text{Set})) \longrightarrow \text{Set}$

# Algebraic effects – ops. and eqs.

**Fibred effect theories**  $\mathcal{T}_{\text{eff}}$ :

- signatures of **dependently typed operation symbols**

$$\frac{\cdot \vdash I \quad x_i : I \vdash O \quad I \text{ and } O \text{ are pure value types}}{\text{op} : (x_i : I) \multimap O}$$

- equipped with **equations** on derivable effect terms

**In eMLTT:**

$$M ::= \dots \mid \text{op}_{\underline{V}}^{\underline{C}}(x.M)$$

**General algebraicity equations** (in addition to **eff. th. eqs.**):

$$\frac{\Gamma \Vdash V : I \quad \Gamma, x : O[V/x_i] \Vdash M : \underline{C} \quad \Gamma \mid \textcolor{red}{z} : \underline{C} \Vdash \textcolor{red}{K} : \underline{D}}{\Gamma \Vdash \textcolor{red}{K}[\textcolor{blue}{\text{op}}_{\underline{V}}^{\underline{C}}(x.M)/\textcolor{red}{z}] = \textcolor{blue}{\text{op}}_{\underline{V}}^{\underline{D}}(x.\textcolor{red}{K}[M/\textcolor{red}{z}]) : \underline{D}} \quad (\text{op} : (x_i : I) \multimap O)$$

**Sound semantics:** Based on families fibrations and Law. theories

- $p : \text{Fam}(\text{Set}) \longrightarrow \text{Set}$  and  $q : \text{Fam}(\text{Mod}(\mathcal{L}_{\mathcal{T}_{\text{eff}}}, \text{Set})) \longrightarrow \text{Set}$

# Algebraic effects – ops. and eqs.

**Fibred effect theories**  $\mathcal{T}_{\text{eff}}$ :

- signatures of **dependently typed operation symbols**

$$\frac{\cdot \vdash I \quad x_i : I \vdash O \quad I \text{ and } O \text{ are pure value types}}{\text{op} : (x_i : I) \multimap O}$$

- equipped with **equations** on derivable effect terms

**In eMLTT:**

$$M ::= \dots \mid \text{op}_{\underline{V}}^{\underline{C}}(x.M)$$

**General algebraicity equations** (in addition to **eff. th. eqs.**):

$$\frac{\Gamma \Vdash V : I \quad \Gamma, x : O[V/x_i] \Vdash M : \underline{C} \quad \Gamma \mid \underline{z} : \underline{C} \Vdash \underline{K} : \underline{D}}{\Gamma \Vdash \underline{K}[\text{op}_{\underline{V}}^{\underline{C}}(x.M)/\underline{z}] = \text{op}_{\underline{V}}^{\underline{D}}(x.\underline{K}[M/\underline{z}]) : \underline{D}} \quad (\text{op} : (x_i : I) \multimap O)$$

**Sound semantics:** Based on **families fibrations** and **Law. theories**

- $\underline{p} : \text{Fam}(\text{Set}) \longrightarrow \text{Set}$  and  $\underline{q} : \text{Fam}(\text{Mod}(\mathcal{L}_{\mathcal{T}_{\text{eff}}}, \text{Set})) \longrightarrow \text{Set}$

# Algebraic effects – examples

## Example 1 (interactive IO):

- $\text{read} : 1 \multimap \text{Chr}$   
 $\text{write} : \text{Chr} \multimap 1$
- no equations

$$(\text{Chr} \stackrel{\text{def}}{=} 1 + \dots + 1)$$

## Example 2 (global state with location-dependent store type):

- $\diamond \vdash \text{Loc}$   
 $\ell : \text{Loc} \vdash \text{Val}$   
 $\diamond \Vdash \text{isDec}_{\text{Loc}} : \prod \ell : \text{Loc} . \prod \ell' : \text{Loc} . (\ell =_{\text{Loc}} \ell') + (\ell =_{\text{Loc}} \ell' \rightarrow 0)$
- $\text{get} : (\ell : \text{Loc}) \multimap \text{Val}$   
 $\text{put} : (\sum \ell : \text{Loc} . \text{Val}) \multimap 1$
- five equations (two of them branching on  $\text{isDec}_{\text{Loc}}$ )

## Example 3 (dep. typed update monads $TX \stackrel{\text{def}}{=} \prod_{s:S} . P s \times X$ )

# Algebraic effects – examples

**Example 1** (interactive IO):

- $\text{read} : 1 \multimap \text{Chr}$   
 $\text{write} : \text{Chr} \multimap 1$
- no equations

$$(\text{Chr} \stackrel{\text{def}}{=} 1 + \dots + 1)$$

**Example 2** (global state with location-dependent store type):

- $\diamond \vdash \text{Loc}$   
 $\ell : \text{Loc} \vdash \text{Val}$   
 $\diamond \Vdash \text{isDec}_{\text{Loc}} : \prod \ell : \text{Loc} . \prod \ell' : \text{Loc} . (\ell =_{\text{Loc}} \ell') + (\ell =_{\text{Loc}} \ell' \rightarrow 0)$
- $\text{get} : (\ell : \text{Loc}) \multimap \text{Val}$   
 $\text{put} : (\sum \ell : \text{Loc} . \text{Val}) \multimap 1$
- five equations (two of them branching on  $\text{isDec}_{\text{Loc}}$ )

**Example 3** (dep. typed update monads  $T X \stackrel{\text{def}}{=} \prod_{s:S} . P s \times X$ )

# Algebraic effects – examples

**Example 1** (interactive IO):

- $\text{read} : 1 \multimap \text{Chr}$   
 $\text{write} : \text{Chr} \multimap 1$
- no equations

$$(\text{Chr} \stackrel{\text{def}}{=} 1 + \dots + 1)$$

**Example 2** (global state with location-dependent store type):

- $\diamond \vdash \text{Loc}$   
 $\ell : \text{Loc} \vdash \text{Val}$   
 $\diamond \Vdash \text{isDec}_{\text{Loc}} : \prod \ell : \text{Loc} . \prod \ell' : \text{Loc} . (\ell =_{\text{Loc}} \ell') + (\ell =_{\text{Loc}} \ell' \rightarrow 0)$
- $\text{get} : (\ell : \text{Loc}) \multimap \text{Val}$   
 $\text{put} : (\sum \ell : \text{Loc} . \text{Val}) \multimap 1$
- five equations (two of them branching on  $\text{isDec}_{\text{Loc}}$ )

**Example 3** (dep. typed update monads  $T X \stackrel{\text{def}}{=} \prod_{s:S} . P s \times X$ )

# **Handlers of algebraic effects**

**(for programming and extrinsic reasoning)**



# Handlers of alg. effects – for programming

**Idea:** Generalisation of exception handlers [Plotkin, Pretnar'09]

Handler  $\sim$  Algebra    and    Handling  $\sim$  Homomorphism

Usual term-level presentation:

$\Gamma \models M$  handled with  $\{\text{op}_{x_v}(x_k) \mapsto N_{\text{op}}\}_{\text{op} \in \mathcal{T}_{\text{eff}}}$  to  $y:A$  in  $\underline{C}$   $N_{\text{ret}} : \underline{C}$

satisfying

$(\text{return } V)$  handled with  $\{\dots\}_{\text{op} \in \mathcal{T}_{\text{eff}}}$  to  $y:A$  in  $N_{\text{ret}} = N_{\text{ret}}[V/x]$

$(\text{op}_V^C(x.M))$  handled with  $\{\dots\}_{\text{op} \in \mathcal{T}_{\text{eff}}}$  to  $y:A$  in  $N_{\text{ret}} = N_{\text{op}}[V/x_v][\dots/x_k]$

Example use case for programming:

- write your programs using alg. ops. (e.g., get and put)
- use handlers to provide fit-for-purpose impl. (e.g.,  $S \rightarrow X \times S$ )

# Handlers of alg. effects – for programming

**Idea:** Generalisation of exception handlers [Plotkin, Pretnar'09]

Handler  $\sim$  Algebra    and    Handling  $\sim$  Homomorphism

**Usual term-level presentation:**

$\Gamma \models M$  handled with  $\{\text{op}_{x_v}(x_k) \mapsto N_{\text{op}}\}_{\text{op} \in \mathcal{T}_{\text{eff}}}$  to  $y:A$  in  $\underline{C}$   $N_{\text{ret}} : \underline{C}$

satisfying

$(\text{return } V)$  handled with  $\{\dots\}_{\text{op} \in \mathcal{T}_{\text{eff}}}$  to  $y:A$  in  $N_{\text{ret}} = N_{\text{ret}}[V/x]$

$(\text{op}_V^C(x.M))$  handled with  $\{\dots\}_{\text{op} \in \mathcal{T}_{\text{eff}}}$  to  $y:A$  in  $N_{\text{ret}} = N_{\text{op}}[V/x_v][\dots/x_k]$

**Example use case for programming:**

- write your programs using alg. ops. (e.g., get and put)
- use handlers to provide fit-for-purpose impl. (e.g.,  $S \rightarrow X \times S$ )

# Handlers of alg. effects – for programming

**Idea:** Generalisation of exception handlers [Plotkin, Pretnar'09]

Handler  $\sim$  Algebra    and    Handling  $\sim$  Homomorphism

**Usual term-level presentation:**

$\Gamma \models M$  handled with  $\{\text{op}_{x_v}(x_k) \mapsto N_{\text{op}}\}_{\text{op} \in \mathcal{T}_{\text{eff}}}$  to  $y:A$  in  $\underline{C}$   $N_{\text{ret}} : \underline{C}$

satisfying

$(\text{return } V)$  handled with  $\{\dots\}_{\text{op} \in \mathcal{T}_{\text{eff}}}$  to  $y:A$  in  $N_{\text{ret}} = N_{\text{ret}}[V/x]$

$(\text{op}_{\underline{V}}^{\underline{C}}(x.M))$  handled with  $\{\dots\}_{\text{op} \in \mathcal{T}_{\text{eff}}}$  to  $y:A$  in  $N_{\text{ret}} = N_{\text{op}}[V/x_v][\dots/x_k]$

**Example use case for programming:**

- write your programs using alg. ops. (e.g., get and put)
- use handlers to provide fit-for-purpose impl. (e.g.,  $S \rightarrow X \times S$ )

# Handlers of alg. effects – for programming

**Idea:** Generalisation of exception handlers [Plotkin, Pretnar'09]

Handler  $\sim$  Algebra    and    Handling  $\sim$  Homomorphism

**Usual term-level presentation:**

$\Gamma \models M$  handled with  $\{\text{op}_{x_v}(x_k) \mapsto N_{\text{op}}\}_{\text{op} \in \mathcal{T}_{\text{eff}}}$  to  $y:A$  in  $\underline{C}$   $N_{\text{ret}} : \underline{C}$

satisfying

$(\text{return } V)$  handled with  $\{\dots\}_{\text{op} \in \mathcal{T}_{\text{eff}}}$  to  $y:A$  in  $N_{\text{ret}} = N_{\text{ret}}[V/x]$

$(\text{op}_{\underline{V}}^{\underline{C}}(x.M))$  handled with  $\{\dots\}_{\text{op} \in \mathcal{T}_{\text{eff}}}$  to  $y:A$  in  $N_{\text{ret}} = N_{\text{op}}[V/x_v][\dots/x_k]$

**Example use case for programming:**

- write your programs using **alg. ops.** (e.g., get and put)
- use handlers to provide **fit-for-purpose impl.** (e.g.,  $S \rightarrow X \times S$ )

# Handlers of alg. effects – for reasoning

**Idea:** Using a derived handle-into-values handling construct

$M$  handled with  $\{\text{op}_{x_v}(x_k) \mapsto V_{\text{op}}\}_{\text{op} \in \mathcal{T}_{\text{eff}}}$  to  $y:A \text{ in}_B V_{\text{ret}}$

we can define natural predicates (essentially, dependent types)

$$\Gamma \Vdash P : UFA \rightarrow \mathcal{U}$$

by

- equipping a universe  $\mathcal{U}$  with an algebra for  $\mathcal{T}_{\text{eff}}$ , and
- using the above handle-into-values construct to define  $P$

**Note 1:**  $P(\text{thunk } M)$  computes a proof obligation for  $M$

**Note 2:** Formally, this is done in an extension of eMLTT with

- a universe  $\mathcal{U}$  closed under  $\text{Nat}$ ,  $1$ ,  $0$ ,  $+$ ,  $\Sigma$ , and  $\Pi$
- a type-based treatment of handlers  $\underline{C} ::= \dots \mid \langle A; \overrightarrow{V_{\text{op}}}; \overrightarrow{W_{\text{eq}}} \rangle$
- function extensionality (actually, it's a bit more extensional)

# Handlers of alg. effects – for reasoning

**Idea:** Using a derived **handle-into-values** handling construct

$M$  handled with  $\{\text{op}_{x_v}(x_k) \mapsto V_{\text{op}}\}_{\text{op} \in \mathcal{T}_{\text{eff}}}$  to  $y:A$  in <sub>$\mathcal{B}$</sub>   $V_{\text{ret}}$

we can define natural **predicates** (essentially, dependent types)

$$\Gamma \Vdash P : UFA \rightarrow \mathcal{U}$$

by

- equipping a **universe**  $\mathcal{U}$  with an **algebra** for  $\mathcal{T}_{\text{eff}}$ , and
- using the above **handle-into-values** construct to define  $P$

**Note 1:**  $P(\text{thunk } M)$  computes a proof obligation for  $M$

**Note 2:** Formally, this is done in an extension of eMLTT with

- a universe  $\mathcal{U}$  closed under  $\text{Nat}$ ,  $1$ ,  $0$ ,  $+$ ,  $\Sigma$ , and  $\Pi$
- a type-based treatment of handlers  $\underline{C} ::= \dots \mid \langle A; \overrightarrow{V_{\text{op}}}; \overrightarrow{W_{\text{eq}}} \rangle$
- function extensionality (actually, it's a bit more extensional)

# Handlers of alg. effects – for reasoning

**Idea:** Using a derived **handle-into-values** handling construct

$M$  handled with  $\{\text{op}_{x_v}(x_k) \mapsto V_{\text{op}}\}_{\text{op} \in \mathcal{T}_{\text{eff}}}$  to  $y:A \text{ in}_B V_{\text{ret}}$

we can define natural **predicates** (essentially, dependent types)

$$\Gamma \Vdash P : UFA \rightarrow \mathcal{U}$$

by

- equipping a **universe**  $\mathcal{U}$  with an **algebra** for  $\mathcal{T}_{\text{eff}}$ , and
- using the above **handle-into-values** construct to define  $P$

**Note 1:**  $P(\text{thunk } M)$  computes a **proof obligation** for  $M$

**Note 2:** Formally, this is done in an extension of eMLTT with

- a universe  $\mathcal{U}$  closed under  $\text{Nat}$ ,  $1$ ,  $0$ ,  $+$ ,  $\Sigma$ , and  $\Pi$
- a type-based treatment of handlers  $\underline{C} ::= \dots \mid \langle A; \overrightarrow{V_{\text{op}}}; \overrightarrow{W_{\text{eq}}} \rangle$
- function extensionality (actually, it's a bit more extensional)

# Handlers of alg. effects – for reasoning

**Idea:** Using a derived **handle-into-values** handling construct

$M$  handled with  $\{\text{op}_{x_v}(x_k) \mapsto V_{\text{op}}\}_{\text{op} \in \mathcal{T}_{\text{eff}}}$  to  $y:A \text{ in}_B V_{\text{ret}}$

we can define natural **predicates** (essentially, dependent types)

$$\Gamma \Vdash P : UFA \rightarrow \mathcal{U}$$

by

- equipping a **universe**  $\mathcal{U}$  with an **algebra** for  $\mathcal{T}_{\text{eff}}$ , and
- using the above **handle-into-values** construct to define  $P$

**Note 1:**  $P(\text{thunk } M)$  computes a **proof obligation** for  $M$

**Note 2:** Formally, this is done in an **extension of eMLTT** with

- a universe  $\mathcal{U}$  closed under  $\text{Nat}$ ,  $1$ ,  $0$ ,  $+$ ,  $\Sigma$ , and  $\Pi$
- a **type-based treatment of handlers**  $\underline{C} ::= \dots \mid \langle A; \overrightarrow{V_{\text{op}}}; \overrightarrow{W_{\text{eq}}} \rangle$
- function extensionality (actually, it's a bit more extensional)



# Handlers of alg. effects – for reasoning

## Example 1 (Evaluation Logic style modalities):

- Given a predicate  $P : A \rightarrow \mathcal{U}$  on return values,  
we define a predicate  $\Diamond P : UFA \rightarrow \mathcal{U}$  on IO-computations as

$$\Diamond P \stackrel{\text{def}}{=} \lambda x : UFA. (\text{force } x) \text{ handled with } \{\dots\}_{\text{op} \in \mathcal{T}_{\text{IO}}} \text{ to } y : A \text{ in } P y$$

using the handler given by

$$V_{\text{read}} \stackrel{\text{def}}{=} \lambda x : (\Sigma x_v : 1. \text{Chr} \rightarrow \mathcal{U}). \widehat{\Sigma} y : \text{El}(\widehat{\text{Chr}}). (\text{snd } x) y$$

$$V_{\text{write}} \stackrel{\text{def}}{=} \lambda x : (\Sigma x_v : \text{Chr}. 1 \rightarrow \mathcal{U}). (\text{snd } x) \star$$

- $\Diamond P$  corresponds to Evaluation Logic's possibility modality

$$\Diamond P (\text{think}(\text{read}(x.\text{write}_{e'}(\text{return } V)))) = \widehat{\Sigma} x : \text{El}(\widehat{\text{Chr}}). P V$$

- To get the necessity modality  $\Box P$ , we use  $\widehat{\Pi} x : \text{El}(\widehat{\text{Chr}})$  in  $V_{\text{read}}$

# Handlers of alg. effects – for reasoning

**Example 1** (Evaluation Logic style modalities):

- Given a predicate  $P : A \rightarrow \mathcal{U}$  on return values,  
we define a predicate  $\Diamond P : UFA \rightarrow \mathcal{U}$  on IO-computations as

$$\Diamond P \stackrel{\text{def}}{=} \lambda x : UFA. (\text{force } x) \text{ handled with } \{\dots\}_{\text{op} \in \mathcal{T}_{\text{IO}}} \text{ to } y : A \text{ in } P y$$

using the handler given by

$$V_{\text{read}} \stackrel{\text{def}}{=} \lambda x : (\sum x_v : 1. \text{Chr} \rightarrow \mathcal{U}). \hat{\Sigma} y : \text{El}(\widehat{\text{Chr}}). (\text{snd } x) y$$

$$V_{\text{write}} \stackrel{\text{def}}{=} \lambda x : (\sum x_v : \text{Chr} . 1 \rightarrow \mathcal{U}). (\text{snd } x) \star$$

- $\Diamond P$  corresponds to Evaluation Logic's possibility modality

$$\Diamond P (\text{think}(\text{read}(x.\text{write}_e(\text{return } V)))) = \hat{\Sigma} x : \text{El}(\widehat{\text{Chr}}). P V$$

- To get the necessity modality  $\Box P$ , we use  $\hat{\Pi} x : \text{El}(\widehat{\text{Chr}})$  in  $V_{\text{read}}$

# Handlers of alg. effects – for reasoning

**Example 1** (Evaluation Logic style modalities):

- Given a predicate  $P : A \rightarrow \mathcal{U}$  on return values,  
we define a predicate  $\Diamond P : UFA \rightarrow \mathcal{U}$  on IO-computations as

$$\Diamond P \stackrel{\text{def}}{=} \lambda x : UFA. (\text{force } x) \text{ handled with } \{\dots\}_{\text{op} \in \mathcal{T}_{\text{IO}}} \text{ to } y : A \text{ in } P y$$

using the handler given by

$$V_{\text{read}} \stackrel{\text{def}}{=} \lambda x : (\sum x_v : 1. \text{Chr} \rightarrow \mathcal{U}). \hat{\Sigma} y : \text{El}(\widehat{\text{Chr}}). (\text{snd } x) y$$

$$V_{\text{write}} \stackrel{\text{def}}{=} \lambda x : (\sum x_v : \text{Chr} . 1 \rightarrow \mathcal{U}). (\text{snd } x) \star$$

- $\Diamond P$  corresponds to Evaluation Logic's **possibility modality**

$$\Diamond P (\text{think}(\text{read}(x.\text{write}_{e'}(\text{return } V)))) = \hat{\Sigma} x : \text{El}(\widehat{\text{Chr}}). P V$$

- To get the necessity modality  $\Box P$ , we use  $\hat{\Pi} x : \text{El}(\widehat{\text{Chr}})$  in  $V_{\text{read}}$

# Handlers of alg. effects – for reasoning

**Example 1** (Evaluation Logic style modalities):

- Given a predicate  $P : A \rightarrow \mathcal{U}$  on return values,  
we define a predicate  $\Diamond P : UFA \rightarrow \mathcal{U}$  on IO-computations as

$$\Diamond P \stackrel{\text{def}}{=} \lambda x : UFA. (\text{force } x) \text{ handled with } \{\dots\}_{\text{op} \in \mathcal{T}_{\text{IO}}} \text{ to } y : A \text{ in } P y$$

using the handler given by

$$V_{\text{read}} \stackrel{\text{def}}{=} \lambda x : (\sum x_v : 1. \text{Chr} \rightarrow \mathcal{U}). \hat{\Sigma} y : \text{El}(\widehat{\text{Chr}}). (\text{snd } x) y$$

$$V_{\text{write}} \stackrel{\text{def}}{=} \lambda x : (\sum x_v : \text{Chr}. 1 \rightarrow \mathcal{U}). (\text{snd } x) \star$$

- $\Diamond P$  corresponds to Evaluation Logic's **possibility modality**

$$\Diamond P (\text{think}(\text{read}(x.\text{write}_{e'}(\text{return } V)))) = \hat{\Sigma} x : \text{El}(\widehat{\text{Chr}}). P V$$

- To get the **necessity modality**  $\Box P$ , we use  $\hat{\Pi} x : \text{El}(\widehat{\text{Chr}})$  in  $V_{\text{read}}$

# Handlers of alg. effects – for reasoning

**Example 2** (Dijkstra's weakest precondition semantics for state):

- Given a postcondition on return values and final states

$$Q : A \rightarrow S \rightarrow \mathcal{U} \quad (S \stackrel{\text{def}}{=} \prod \ell : \text{Loc} . \text{Val})$$

we define a precondition for stateful comps. on initial states

$$\text{wp}_Q : \text{UFA} \rightarrow S \rightarrow \mathcal{U}$$

by

- 1) handling the given comp. into a state-passing function using

$$V_{\text{get}}, V_{\text{put}} \quad \text{on} \quad S \rightarrow (\mathcal{U} \times S) \quad \text{and} \quad V_{\text{ret}} \text{ " = " } Q$$

- 2) feeding in the initial state; and 3) projecting out  $\mathcal{U}$

- Theorem:**  $\text{wp}_Q$  satisfies expected properties of WPs, e.g.,

$$\text{wp}_Q (\text{thunk}(\text{return } V)) = \lambda x_S : S . Q \ V \ x_S$$

$$\text{wp}_Q (\text{thunk}(\text{put}_{\langle \ell, V \rangle}(M))) = \lambda x_S : S . \text{wp}_Q (\text{thunk } M) (x_S[\ell \mapsto V])$$

# Handlers of alg. effects – for reasoning

**Example 2** (Dijkstra's weakest precondition semantics for state):

- Given a postcondition on **return values** and **final states**

$$Q : A \rightarrow S \rightarrow \mathcal{U} \quad (S \stackrel{\text{def}}{=} \prod \ell : \text{Loc} . \text{Val})$$

we define a precondition for **stateful comps.** on **initial states**

$$\text{wp}_Q : \text{UFA} \rightarrow S \rightarrow \mathcal{U}$$

by

- 1) handling the given comp. into a **state-passing function** using

$$V_{\text{get}}, V_{\text{put}} \quad \text{on} \quad S \rightarrow (\mathcal{U} \times S) \quad \text{and} \quad V_{\text{ret}} \text{ “=” } Q$$

- 2) feeding in the **initial state**; and 3) **projecting** out  $\mathcal{U}$

- Theorem:**  $\text{wp}_Q$  satisfies expected properties of WPs, e.g.,

$$\text{wp}_Q (\text{thunk}(\text{return } V)) = \lambda x_S : S . Q \ V \ x_S$$

$$\text{wp}_Q (\text{thunk}(\text{put}_{\langle \ell, V \rangle}(M))) = \lambda x_S : S . \text{wp}_Q (\text{thunk } M) (x_S[\ell \mapsto V])$$

# Handlers of alg. effects – for reasoning

**Example 2** (Dijkstra's weakest precondition semantics for state):

- Given a postcondition on **return values** and **final states**

$$Q : A \rightarrow S \rightarrow \mathcal{U} \quad (S \stackrel{\text{def}}{=} \prod \ell : \text{Loc} . \text{Val})$$

we define a precondition for **stateful comps.** on **initial states**

$$\text{wp}_Q : \text{UFA} \rightarrow S \rightarrow \mathcal{U}$$

by

- 1) handling the given comp. into a **state-passing function** using

$$V_{\text{get}}, V_{\text{put}} \quad \text{on} \quad S \rightarrow (\mathcal{U} \times S) \quad \text{and} \quad V_{\text{ret}} \text{ “=” } Q$$

- 2) feeding in the **initial state**; and 3) **projecting** out  $\mathcal{U}$

- Theorem:**  $\text{wp}_Q$  satisfies expected properties of WPs, e.g.,

$$\text{wp}_Q (\text{thunk}(\text{return } V)) = \lambda x_S : S . Q \text{ } V \text{ } x_S$$

$$\text{wp}_Q (\text{thunk}(\text{put}_{\langle \ell, V \rangle}(M))) = \lambda x_S : S . \text{wp}_Q (\text{thunk } M) (x_S[\ell \mapsto V])$$

# Handlers of alg. effects – for reasoning

## Example 3 (Patterns of allowed (IO-)effects):

- Assuming an inductive type of IO-protocols, given by

$$e : \text{Protocol} \quad r : (\text{Chr} \rightarrow \text{Protocol}) \rightarrow \text{Protocol}$$

$$w : (\text{Chr} \rightarrow \mathcal{U}) \rightarrow \text{Protocol} \rightarrow \text{Protocol}$$

and potentially also by  $\wedge, \vee, \dots$

- Then, we define the predicate (rel. between comps. and protocols)

$$\text{Allowed} : \text{UFA} \rightarrow \text{Protocol} \rightarrow \mathcal{U}$$

by handling the given computation using

$$V_{\text{read}}, V_{\text{write}} \quad \text{on} \quad \text{Protocol} \rightarrow \mathcal{U}$$

where

$$V_{\text{read}} \langle -, V_{rk} \rangle (r \text{ Pr}') \stackrel{\text{def}}{=} \widehat{\Pi} x : \text{El}(\widehat{\text{Chr}}) . (V_{rk} \ x) \ (Pr' \ x)$$

$$V_{\text{write}} \langle V, V_{wk} \rangle (w \ P \ Pr') \stackrel{\text{def}}{=} \widehat{\Sigma} x : \text{El}(P \ V) . V_{wk} \ \star \ Pr'$$

$$\stackrel{\text{def}}{=} \widehat{0}$$



# Handlers of alg. effects – for reasoning

**Example 3** (Patterns of allowed (IO-)effects):

- Assuming an inductive type of **IO-protocols**, given by

$$e : \text{Protocol} \quad r : (\text{Chr} \rightarrow \text{Protocol}) \rightarrow \text{Protocol}$$

$$w : (\text{Chr} \rightarrow \mathcal{U}) \rightarrow \text{Protocol} \rightarrow \text{Protocol}$$

and potentially also by  $\wedge, \vee, \dots$

- Then, we define the predicate (rel. between comps. and protocols)

$$\text{Allowed} : \text{UFA} \rightarrow \text{Protocol} \rightarrow \mathcal{U}$$

by handling the given computation using

$$V_{\text{read}}, V_{\text{write}} \quad \text{on} \quad \text{Protocol} \rightarrow \mathcal{U}$$

where

$$V_{\text{read}} \langle -, V_{rk} \rangle (r \text{ Pr}') \stackrel{\text{def}}{=} \widehat{\Pi} x : \text{El}(\widehat{\text{Chr}}) . (V_{rk} \ x) (Pr' \ x)$$

$$V_{\text{write}} \langle V, V_{wk} \rangle (w \ P \ Pr') \stackrel{\text{def}}{=} \widehat{\Sigma} x : \text{El}(P \ V) . V_{wk} \ \star \ Pr'$$

$$\stackrel{\text{def}}{=} \widehat{0}$$

# Handlers of alg. effects – for reasoning

**Example 3** (Patterns of allowed (IO-)effects):

- Assuming an inductive type of **IO-protocols**, given by

$$\mathbf{e} : \text{Protocol} \quad \mathbf{r} : (\text{Chr} \rightarrow \text{Protocol}) \rightarrow \text{Protocol}$$

$$\mathbf{w} : (\text{Chr} \rightarrow \mathcal{U}) \rightarrow \text{Protocol} \rightarrow \text{Protocol}$$

and potentially also by  $\wedge, \vee, \dots$

- Then, we define the **predicate** (rel. between **comps.** and **protocols**)

$$\text{Allowed} : \text{UFA} \rightarrow \text{Protocol} \rightarrow \mathcal{U}$$

by handling the given computation using

$$V_{\text{read}}, V_{\text{write}} \quad \text{on} \quad \text{Protocol} \rightarrow \mathcal{U}$$

where

$$V_{\text{read}} \langle -, V_{\text{rk}} \rangle (\mathbf{r} \text{ Pr}') \stackrel{\text{def}}{=} \widehat{\Pi} x : \text{El}(\widehat{\text{Chr}}) . (V_{\text{rk}} x) (\text{Pr}' x)$$

$$V_{\text{write}} \langle V, V_{\text{wk}} \rangle (\mathbf{w} P \text{ Pr}') \stackrel{\text{def}}{=} \widehat{\Sigma} x : \text{El}(P V) . V_{\text{wk}} \star \text{Pr}'$$

$$\text{—} \stackrel{\text{def}}{=} \widehat{0}$$

# Conclusion

At a high-level, the presented work was about combining  
dependent types and computational effects

In particular, you saw

- a clean core language of dependent types and comp. effects
- a natural category-theoretic semantics
- alg. effects and handlers, in particular, for reasoning using
  - Evaluation Logic style modalities
  - Dijkstra's weakest precondition semantics for state
  - patterns of allowed (IO-)effects

Ongoing and future work:

- uniform account of the various handler-defined predicates
- more expressive comp. types (par. adjunctions, Dijkstra monads)
- type-dependency on computations (e.g., in seq. composition)

# Thank you!

D. Ahman.

**Fibred Computational Effects.** (PhD Thesis, 2017)

D. Ahman, N. Ghani, G. Plotkin.

**Dependent Types and Fibred Computational Effects.** (FoSSaCS'16)

D. Ahman.

**Handling Fibred Computational Effects.** (POPL'18)