A fibrational view on computational effects

Danel Ahman
Prosecco Team, Inria Paris

Ljubljana, 31 May 2018
Background – dependent types

The Curry-Howard correspondence:

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

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

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

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

which in turn could be used for typing sorting functions

$$\forall \exists \text{ sort : } \Pi \ell : (\text{List Nat}) . \Sigma \ell' : (\text{List Nat}) . \left( \text{Sorted}(\ell') \times \ldots \right)$$

Large examples: CompCert (Coq), miTLS and HACL* (F*), \ldots
Background – dependent types

The Curry-Howard correspondence:

Simple Types  \sim Propositional Logic  \quad (\text{Nat, String, \ldots})

Dependent Types  \sim Predicate Logic  \quad (\Sigma, \Pi, =, \ldots)

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

\[ \ell : (\text{List Nat}) \vdash \text{Sorted}(\ell) \quad \overset{\text{def}}{=} \quad \Pi i: \text{Nat}. (0 < i < \text{len } \ell) \rightarrow (\ell[i-1] \leq \ell[i]) \]

which in turn could be used for typing sorting functions

\[ \forall \exists \wedge \sigma r : \Pi \ell : (\text{List Nat}). \Sigma \ell' : (\text{List Nat}). \left( \text{Sorted}(\ell') \times \ldots \right) \]

Large examples: CompCert (Coq), miTLS and HACL* (F*), \ldots
Background – dependent types

The Curry-Howard correspondence:

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

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

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

\begin{align*}
\forall \ell : (\text{List Nat}) \vdash \text{Sorted}(\ell) \overset{\text{def}}{=} \Pi i : \text{Nat}. (0 < i < \text{len } \ell) \rightarrow (\ell[i-1] \leq \ell[i])
\end{align*}

which in turn could be used for typing sorting functions

\begin{align*}
\forall \exists \wedge
\text{sort} : \Pi \ell : (\text{List Nat}) \cdot \Sigma \ell' : (\text{List Nat}) \cdot (\text{Sorted}(\ell') \times \ldots)
\end{align*}

Large examples: CompCert (Coq), miTLS and HACL* (F*), \ldots
Background – computational effects

Examples:

- state, exceptions, divergence, IO, nondeterminism, probability, . . .

Meta-languages and models for comp. effects: based on

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

\[
\Gamma \vdash M : A \xrightarrow{\lambda_c} [\Gamma] \rightarrow T[A]
\]

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

\[
\Gamma \vdash V : A_{CBPV} : [\Gamma] \rightarrow [A] \quad \Gamma \vdash M : C_{CBPV} : [\Gamma] \rightarrow U([C])
\]

- algebraic presentations (Plotkin and Power)

\[
\text{get} : 1 \rightarrow S \quad \text{put} : S \rightarrow 1 \quad (+ \text{ equations})
\]
Outline – putting the two together

We investigate the combination of

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

Goals

- tell a mathematically natural story
- use established math. techniques
- cover a wide range of comp. effects
- discover smth. interesting
Outline – putting the two together

We investigate the combination of

- dependent types \( \Pi, \Sigma, V =_A W, \ldots \)
- computational effects (state, nondeterminism, IO, ...)

Goals

- tell a mathematically natural story (via a clean core calculus)
- use established math. techniques
- cover a wide range of comp. effects
- discover smth. interesting
Outline – putting the two together

We investigate the combination of

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

Goals

- tell a mathematically natural story (via a clean core calculus)
- use established math. techniques (fibrations and adjunctions)
- cover a wide range of comp. effects
- discover smth. interesting
Outline – putting the two together

We investigate the combination of

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

Goals

- tell a mathematically natural story \((\text{via a clean core calculus})\)
- use established math. techniques \((\text{fibrations and adjunctions})\)
- cover a wide range of comp. effects \((\text{alg. effects, continuations})\)
- discover smth. interesting
Outline – putting the two together

We investigate the combination of

- dependent types \((\Pi, \Sigma, V \Rightarrow A W, \ldots)\)
- computational effects (state, nondeterminism, IO, ...)

Goals

- tell a mathematically natural story (via a clean core calculus)
- 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)
Outline – putting the two together

We investigate the combination of

- dependent types \((\Pi, \Sigma, V =_A W, ...\)\)
- computational effects (state, nondeterminism, IO, ...)

Goals

- tell a mathematically natural story (via a clean core calculus)
- 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)

Two guiding problems

- effectful programs in types (e.g., get and put in types)
- typing of effectful programs (e.g., sequential composition)
Effectful programs in types
(type-dependency in the presence of effects)
Effectful programs in types

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

A1: In this work, we say not directly

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

A2: Various people are also looking at the direct case

- type-dependency needs to be “homomorphic”
- intuitively,
  - need to lift Sorted(\ell) to Sorted^\dagger(c), where \( c : T(\text{List Chr}) \)
  
  \[
  \text{Sorted}^\dagger(\text{receive(y. return y)}) = \langle \text{receive} \rangle(y. \text{Sorted(y)})
  \]

- for this Sorted needs to be a \( T \)-algebra
- (cf. recent papers by Pédrot and Tabareau; Bowman et al.)
Effectful programs in types

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

A1: In this work, we say not directly  
   - types should only depend on static information about effects  
   - allow dependency on effectful comps. via analysing thunks

A2: Various people are also looking at the direct case  
   - type-dependency needs to be “homomorphic”  
   - intuitively,  
     - need to lift Sorted(\(\ell\)) to Sorted\(^\dagger\)(c), where c: T(List Chr)  
       \[
       \text{Sorted}^\dagger(\text{receive}(y.\text{return}\ y)) = \langle \text{receive} \rangle(\text{receive}(y.\text{Sorted}(y)))
       \]
     - for this Sorted needs to be a \(T\)-algebra  
   - (cf. recent papers by Pédrot and Tabareau; Bowman et al.)
Effectful programs in types

Q: Should we allow situations such as Sorted[receive(y. M)/ℓ]?

A1: In this work, we say not directly
- types should only depend on static information about effects
- allow dependency on effectful comps. via analysing thunks

A2: Various people are also looking at the direct case
- type-dependency needs to be “homomorphic”
- intuitively,
  - need to lift Sorted(ℓ) to Sorted†(c), where c : T(List Chr)
    \[
    \text{Sorted}^\dagger(\text{receive}(y. \text{return } y)) = \langle \text{receive} \rangle(y. \text{Sorted}(y))
    \]
  - for this Sorted needs to be a \( T \)-algebra
- (cf. recent papers by Pédrot and Tabareau; Bowman et al.)
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 $\Gamma \vdash C$ (dep. typed CBPV/EEC)
- where $\Gamma$ contains only value variables $x_1 : A_1, \ldots, x_n : A_n$

Could have also considered Moggi’s $\lambda_{ML}$ or Levy’s FGCBV
- building on CBPV/EEC gives a more general story
- especially for the treatment of sequential composition
- and also for (Idris-style parameterised) dependent effect-typing
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 $\Gamma \vdash C$ (dep. typed CBPV/EEC)
- where $\Gamma$ contains only value variables $x_1 : A_1, \ldots, x_n : A_n$

Could have also considered Moggi’s $\lambda_{ML}$ or Levy’s FGCBV

- building on CBPV/EEC gives a more general story
- especially for the treatment of sequential composition
- and also for (Idris-style parameterised) dependent effect-typing
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 $\Gamma \vdash C$ (dep. typed CBPV/EEC)
- where $\Gamma$ contains only value variables $x_1 : A_1, \ldots, x_n : A_n$

Could have also considered Moggi’s $\lambda_{ML}$ or Levy’s FGCBV

- building on CBPV/EEC gives a more general story
- especially for the treatment of sequential composition
- and also for (Idris-style parameterised) dependent effect-typing
Typing of effectful programs
(e.g., sequential composition)
Assigning types to effectful programs

The problem: The standard typing rule for seq. composition

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

is not correct any more because it potentially allows

\[x \in \text{FV}(C)\]

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 $C$: [Levy'04]

\[
\Gamma \vdash M : F A \quad \Gamma \vdash C \quad \Gamma, x:A \vdash N : C \\
\Gamma \vdash M \text{ to } x:A \text{ in } N : C
\]

**But:** Sometimes it is useful if $C$ can depend on $x$!

- say we consider

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

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

  \[
  x:\text{Bool} \vdash C(x) \overset{\text{def}}{=} \text{if } x \text{ then “allow fread, fwrite, and fclose” else “allow fopen”}
  \]

  (needs more expressive comp. types than in the core calculus)
Assigning types to effectful programs

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

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

$$
\Gamma \vdash c \quad \Gamma, x:A \vdash N : C

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

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

- say we consider

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

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

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

(needs more expressive comp. types than in the core calculus)
Assigning types to effectful programs

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

**Option 1:** We could restrict the free variables in $\mathcal{C}$:  

\[
\frac{\Gamma \vdash M : F \ A \quad \Gamma \vdash \mathcal{C} \quad \Gamma, x:A \vdash N : \mathcal{C}}{
\Gamma \vdash M \ \text{to} \ x:A \ \text{in} \ N : \mathcal{C}}
\]

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

- say we consider

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

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

\[
x: \text{Bool} \vdash \mathcal{C}(x) \overset{\text{def}}{=} \text{if } x \text{ then “allow fread, fwrite, and fclose”}
\]

\[
\text{else “allow fopen”}
\]

(needs more expressive comp. types than in the core calculus)
Assigning types to effectful programs

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

Option 2: One could lift sequential composition to type level

\[ \Gamma \vdash M \text{ to } x : A \text{ in } N : M \text{ to } x : A \text{ in } C \]

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

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

\[
\begin{align*}
\Gamma & \vdash M : T A \\
\Gamma, x : A & \vdash N : T B(x) \\
\hline
\Gamma & \vdash M \text{ to } x : A \text{ in } N : T (\Sigma x : A.B)
\end{align*}
\]

**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

\[ \Gamma \vdash M \text{ to } x : A \text{ in } N : M \text{ to } x : A \text{ in } C \]

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

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

\[ \Gamma \vdash M : T \ A \quad \Gamma, x : A \vdash N : T \ B(x) \]

\[ \Gamma \vdash M \text{ to } x : A \text{ in } N : T (\Sigma x : A.B) \]

**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

\[ \Gamma \vdash M \text{ to } x:A \text{ in } N : M \text{ to } x:A \text{ in } C \]

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

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

\[
\begin{align*}
\Gamma & \vdash M : T \ A \\
\Gamma, x:A & \vdash N : T \ B(x) \\
\Gamma & \vdash M \text{ to } x:A \text{ in } N : T \ (\Sigma x : A.B)
\end{align*}
\]

**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

**Our solution:** We draw inspiration from algebraic effects
- and combine this with restricting $C$ in seq. comp. (Option 1)

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

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

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

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

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

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

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*

**Our solution:** We draw inspiration from *algebraic effects*

- and combine this with restricting $C$ in seq. comp. (Option 1)

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

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

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

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

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

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

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**

**Our solution:** We draw inspiration from **algebraic effects**

- and combine this with restricting $C$ in seq. comp. (Option 1)

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

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

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

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

**Idea:** \( M \) denotes an element of the coproduct of algebras

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

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

**Our solution:** We draw inspiration from algebraic effects

- and combine this with restricting $C$ in seq. comp. (Option 1)

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

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

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

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

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

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

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

**Our solution:** We draw inspiration from algebraic effects

- and combine this with restricting $C$ in seq. comp. (Option 1)

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

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

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

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

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

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

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

eMLTT: a core dep.-typed calculus with comp. effects
eMLTT – value and comp. types

Value types: MLTT + thunks + …

\[ A, B ::= \text{Nat} \mid 1 \mid 0 \mid \Pi x : A. B \mid \Sigma x : A. B \mid V =_A W \mid UC \mid \ldots \]

- \( UC \) is the type of thunked (i.e., suspended) computations

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

\[ C, D ::= FA \mid \Pi x : A. C \mid \Sigma x : A. C \]

- \( FA \) is the type of computations returning values of type \( A \)
- \( \Pi x : A. C \) is the type of dependent effectful functions
  - generalises CBPV/EEC’s comp. types \( A \to C \) and \( C \times D \)
- \( \Sigma x : A. C \) 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 \( !A \otimes C \) and \( C \oplus D \)
eMLTT – value and comp. types

Value types: MLTT + thunks + ...

\[ A, B ::= \text{Nat} \mid 1 \mid 0 \mid \Pi x:A. B \mid \Sigma x:A. B \mid V =_A W \mid UC \mid \ldots \]

- \(UC\) is the type of thunked (i.e., suspended) computations

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

\[ C, D ::= FA \mid \Pi x:A. C \mid \Sigma x:A. C \]

- \(FA\) is the type of computations returning values of type \(A\)
- \(\Pi x:A. C\) is the type of dependent effectful functions
  - generalises CBPV/EEC’s comp. types \(A \to C\) and \(C \times D\)
- \(\Sigma x:A. C\) 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 \(!A \otimes C\) and \(C \oplus D\)
Value terms: MLTT + thunks + ...

\[ V, W ::= x \mid \text{zero} \mid \text{succ} \ V \mid \ldots \mid \text{thunk} \ M \mid \ldots \]

- equational theory based on intensional MLTT

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

\[ M, N ::= \text{force} \ V \]

\[ \quad | \text{return} \ V \]

\[ \quad | M \text{ to } x:A \text{ in } N \]

\[ \quad | \lambda x:A. M \]

\[ \quad | M \langle V \rangle \]

\[ \quad | M \text{ to } \langle x:A, z:C \rangle \text{ in } K \quad \text{(comp. } \Sigma \text{ intro.)} \]

\[ \quad | M \text{ to } \langle x:A, z:C \rangle \text{ in } K \quad \text{(comp. } \Sigma \text{ elim.)} \]

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

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

\[ V, W ::= x \mid \text{zero} \mid \text{succ } V \mid \ldots \mid \text{thunk } M \mid \ldots \]

- equational theory based on intensional MLTT

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

\[ M, N ::= \text{force } V \]

\[ \quad \mid \text{return } V \]

\[ \quad \mid M \text{ to } x:A \text{ in } N \]

\[ \quad \mid \lambda x:A. M \]

\[ \quad \mid MV \]

\[ \quad \mid \langle V, M \rangle \quad \text{(comp. } \Sigma \text{ intro.)} \]

\[ \quad \mid M \text{ to } \langle x:A, z:C \rangle \text{ in } K \quad \text{(comp. } \Sigma \text{ elim.)} \]

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

\[ V, W ::= x \mid \text{zero} \mid \text{succ } V \mid \ldots \mid \text{thunk } M \mid \ldots \]

- equational theory based on intensional MLTT

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

\[ M, N ::= \text{force } V \]
\[ \mid \text{return } V \]
\[ \mid M \text{ to } x:A \text{ in } N \]
\[ \mid \lambda x:A.M \]
\[ \mid MV \]
\[ \mid \langle V, M \rangle \]
\[ \mid M \text{ to } \langle x:A, z:C \rangle \text{ in } K \]

(comp. Σ intro.)
(comp. Σ elim.)

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

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

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

$$K, L ::= z \quad \quad \quad \quad \quad \quad \text{(linear comp. vars.)}$$

$$\mid K \text{ to } x : A \text{ in } M$$

$$\mid \lambda x : A . K$$

$$\mid KV$$

$$\mid \langle V, K \rangle \quad \quad \quad \quad \quad \quad \text{(comp. } \Sigma \text{ intro.)}$$

$$\mid K \text{ to } \langle x : A, z : C \rangle \text{ in } L \quad \quad \quad \quad \quad \quad \text{(comp. } \Sigma \text{ elim.)}$$

**Typing judgments:**

- $\Gamma \vdash V : A$
- $\Gamma \vdash M : C$
- $\Gamma \mid z : C \vdash K : D$ \quad \text{(linear in } z; \text{ comp. bound to } z \text{ 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

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

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

$$K, L ::= z \quad \text{(linear comp. vars.)}$$

<table>
<thead>
<tr>
<th>$K$ to $x : A$ in $M$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda x : A . K$</td>
</tr>
<tr>
<td>$KV$</td>
</tr>
<tr>
<td>$\langle V, K \rangle$</td>
</tr>
<tr>
<td>$K$ to $\langle x : A, z : C \rangle$ in $L$ \hspace{1cm} \text{(comp. } \Sigma \text{ elim.)}$</td>
</tr>
</tbody>
</table>

Typing judgments:

- $\Gamma \vdash V : A$
- $\Gamma \vdash M : C$
- $\Gamma \mid z : C \mid h \ K : D$ \hspace{1cm} \text{(linear in } z; \text{ comp. bound to } z \text{ happens first)}
We can then account for type-dependency in seq. comp. as

\[
\begin{align*}
\Gamma \vdash c \; M : F A \\
\Gamma \vdash \Sigma x : A . \_C(x) \\
\Gamma \vdash c \; \langle x, N \rangle : \Sigma x : A . \_C(x)
\end{align*}
\]

\[
\Gamma \vdash M \; \text{to} \; x : A \; \text{in} \; \langle x, N \rangle : \Sigma x : A . \_C(x)
\]

As a bonus, the comp. \( \Sigma \)-type can also be used to explain Idris's

\[
\begin{align*}
\Gamma \vdash \varepsilon_1 : \text{Effect} \\
\Gamma \vdash A \Rightarrow \varepsilon_2 : A \rightarrow \text{Effect}
\end{align*}
\]

\[
\Gamma \vdash T \varepsilon_1 A \varepsilon_2
\]

in terms of standard parameterised effect-typing as

\[
T \varepsilon_1 A \varepsilon_2 \overset{\text{def}}{=} U_{\varepsilon_1} (\Sigma x : A . F_{\varepsilon_2} x 1)
\]

and thus naturally accommodate examples like

\[
\text{fopen (return true, return false) to } x : \text{Bool in } N
\]
We can then account for type-dependency in seq. comp. as

\[ \Gamma, x : A \vdash N : C(x) \]
\[ \Gamma \vdash M : F A \]
\[ \Gamma \vdash \Sigma x : A. C(x) \]
\[ \Gamma, x : A \vdash \langle x, N \rangle : \Sigma x : A. C(x) \]
\[ \Gamma \vdash M \text{ to } x : A \text{ in } \langle x, N \rangle : \Sigma x : A. C(x) \]

As a bonus, the comp. \( \Sigma \)-type can also be used to explain Idris’s

\[ \Gamma \vdash \varepsilon_1 : \text{Effect} \quad \Gamma \vdash A \quad \Gamma \vdash \varepsilon_2 : A \rightarrow \text{Effect} \]
\[ \Gamma \vdash T \varepsilon_1 A \varepsilon_2 \]

in terms of standard parameterised effect-typing as

\[ T \varepsilon_1 A \varepsilon_2 \overset{\text{def}}{=} U_{\varepsilon_1} (\Sigma x : A. F_{\varepsilon_2 x} 1) \]

and thus naturally accommodate examples like

\[ \text{fopen (return true, return false) to } x : \text{Bool in } N \]
eMLTT – typing sequential composition

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

\[ \Gamma, x : A \vdash \text{c} M : F A \]
\[ \Gamma \vdash \Sigma x : A . \text{C}(x) \]
\[ \Gamma, x : A \vdash \langle x, N \rangle : \Sigma x : A . \text{C}(x) \]
\[ \Gamma \vdash \text{c} \text{M} \text{to} x : A \text{in} \langle x, N \rangle : \Sigma x : A . \text{C}(x) \]

- As a bonus, the comp. Σ-type can also be used to explain Idris’s

\[ \Gamma \vdash \varepsilon_1 : \text{Effect} \]
\[ \Gamma \vdash A \]
\[ \Gamma \vdash \varepsilon_2 : A \rightarrow \text{Effect} \]
\[ \Gamma \vdash T \varepsilon_1 A \varepsilon_2 \]

in terms of standard parameterised effect-typing as

\[ T \varepsilon_1 A \varepsilon_2 \overset{\text{def}}{=} U_{\varepsilon_1} (\Sigma x : A . F_{\varepsilon_2} x 1) \]

and thus naturally accommodate examples like

\[ \text{fopen (return true, return false)} \text{ to } x : \text{Bool in } N \]
Fibred adjunction models
(categorical semantics of eMLTT)
Fibred adjunction models – value part

Given by a split closed comprehension category \( p \), as in

\[
\begin{array}{c}
\mathcal{V} \\
\downarrow \mathcal{B} \\
\end{array}
\]

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 \( \llbracket \Gamma ; V \rrbracket : \mathbb{1}_{\llbracket \Gamma \rrbracket} \rightarrow A \) in \( \mathcal{V}_{\llbracket \Gamma \rrbracket} \)
Fibred adjunction models – value part

Given by a split closed comprehension category \( p \), as in

\[
\begin{array}{c}
\mathcal{V} \\
\downarrow p \\
\downarrow 1 \\
\{ - \} \\
\downarrow \mathcal{B}
\end{array}
\]

such that

- \( p \) has split fibred strong colimits of shape 0 and 2 \([Jacobs’99]\)
  - (in thesis, also Jacobs-style characterisation 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

![Diagram](image)

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

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

\[
\begin{array}{ccc}
    \mathcal{V} & \xleftarrow{\perp} & \mathcal{C} \\
    \downarrow & \quad & \downarrow \\
    \mathcal{B} & \xleftarrow{q} & \mathcal{C}
\end{array}
\]

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 a weak form of fib. enrich. in \( p \) (hom. function type \( \to \))
Fibred adjunction models – correctness

**Theorem (Soundness):**

- If $\Gamma \vdash C$, then $[[\Gamma; C]] \in \mathcal{C}_{[[\Gamma]]}$
- If $\Gamma \vdash_{\mathcal{C}} M : C$, then $[[\Gamma; M]] : 1_{[[\Gamma]]} \rightarrow U([[\Gamma; C]])$
- If $\Gamma \mid z : C \vdash_{\mathcal{C}} K : D$, then $[[\Gamma; z : C; K]] : [[\Gamma; C]] \rightarrow [[\Gamma; D]]$
- If $\Gamma \vdash C = D$, then $[[\Gamma; C]] = [[\Gamma; D]] \in \mathcal{C}_{[[\Gamma]]}$
- ... 

**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

\[
\begin{array}{c}
\mathcal{V} \\
\downarrow^p \quad \downarrow^1 \\
\mathcal{B} \\
\uparrow^q
\end{array}
\quad \quad
\begin{array}{c}
\downarrow^\perp \\
\Rightarrow
\end{array}
\quad \quad
\begin{array}{c}
\mathcal{C} \\
\downarrow^F \\
\Rightarrow
\end{array}
\]
Examples of fibred adjunction models

Example 1 (identity adjunctions):

- sound as long as no actual comp. effects in the calculus

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

- given an adj. model of EEC $F \dashv U : C \to \mathcal{V}$ ($\mathcal{V}$ a CCC, ...), we can lift it to simple fibrations $\hat{F} \dashv \hat{U} : s(\mathcal{V}, C) \to s(\mathcal{V})$

where

$$s_{\mathcal{V}, C} : s(\mathcal{V}, C) \to \mathcal{V}$$

is defined as

$$s_{\mathcal{V}, C} \left( X \in \mathcal{V} , C \in C \right) \overset{\text{def}}{=} X$$

$$s_{\mathcal{V}, C} \left( f : X \to Y , h : X \otimes C \to D \right) \overset{\text{def}}{=} f : s_{\mathcal{V}, C}(X, C) \to s_{\mathcal{V}, C}(Y, D)$$

- doesn’t support any real type dependency (constant families)
Examples of fibred adjunction models

**Example 1** (identity adjunctions):
- sound as long as no actual comp. effects in the calculus

**Example 2** (simple fibrations from enriched adj. models of EEC):
- given an adj. model of EEC \( F \dashv U : \mathcal{C} \to \mathcal{V} \) (\( \mathcal{V} \) a CCC, …),
we can lift it to simple fibrations \( \widehat{F} \dashv \widehat{U} : s(\mathcal{V}, \mathcal{C}) \to s(\mathcal{V}) \)
where

\[
s_{\mathcal{V}, \mathcal{C}} : s(\mathcal{V}, \mathcal{C}) \to \mathcal{V} \]

is defined as

\[
s_{\mathcal{V}, \mathcal{C}} \left( X \in \mathcal{V}, \_ \in \mathcal{C} \right) \overset{\text{def}}{=} X
\]

\[
s_{\mathcal{V}, \mathcal{C}} \left( f : X \to Y, h : X \otimes \_ \to D \right) \overset{\text{def}}{=} f : s_{\mathcal{V}, \mathcal{C}}(X, \_ \in \mathcal{C}) \to s_{\mathcal{V}, \mathcal{C}}(Y, D)
\]

- doesn’t support any real type dependency (constant families)
Examples of fibred adjunction models

Example 3 (families fibrations and lifting of adjunctions):
- given a suitable adjunction $F_{\mathcal{D}} \dashv U_{\mathcal{D}} : \mathcal{D} \to \text{Set}$,
  we can lift it to $\hat{F}_{\mathcal{D}} \dashv \hat{U}_{\mathcal{D}} : \text{Fam}(\mathcal{D}) \to \text{Fam}(\text{Set})$
  between
    $$\text{fam}_{\text{Set}} : \text{Fam}(\text{Set}) \to \text{Set}$$
    $$\text{fam}_{\mathcal{D}} : \text{Fam}(\mathcal{D}) \to \text{Set}$$
- resulting in
  - $[\Gamma; A] = ([\Gamma], [A]) \in \text{Fam}(\text{Set})$ (where $[\Gamma] \in \text{Set}, [A] \in [\Gamma] \to \text{Set}$)
  - $[\Gamma; C] = ([\Gamma], [C]) \in \text{Fam}(\mathcal{D})$ (where $[C] \in [\Gamma] \to \mathcal{D}$)
- examples
  - $F^T \dashv U^T : \text{Set}^T \to \text{Set}$
  - $(-) \times S \dashv (-)^S : \text{Set} \to \text{Set}$
  - $R(-) \dashv R(-) : \text{Set}^{\text{op}} \to \text{Set}$
Examples of fibred adjunction models

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

• given the EM-adjunction $F^T \dashv U^T : \text{CPO}^T \rightarrow \text{CPO}$, we can lift it to $\widehat{F_D} \dashv \widehat{U_D} : \text{CFam}(\text{CPO}^T) \rightarrow \text{CFam}(\text{CPO})$

between

\[
\text{cfam}_{\text{CPO}} : \text{CFam}(\text{CPO}) \rightarrow \text{CPO} \\
\text{cfam}_{\text{CPO}^T} : \text{CFam}(\text{CPO}^T) \rightarrow \text{CPO}
\]

• resulting in

\[
\begin{align*}
& (\llbracket \Gamma \rrbracket, \llbracket A \rrbracket) \in \text{CFam}(\text{CPO}) \quad (\llbracket \Gamma \rrbracket \in \text{CPO}, \llbracket A \rrbracket \in \llbracket \Gamma \rrbracket \rightarrow \text{CPO}^{EP}) \\
& (\llbracket \Gamma \rrbracket, \llbracket C \rrbracket) \in \text{CFam}(\text{CPO}^T) \quad (\llbracket C \rrbracket \in \llbracket \Gamma \rrbracket \rightarrow (\text{CPO}^T)^{EP})
\end{align*}
\]

• if $T$ supports a least zero-ary op., then it also models recursion

\[
M ::= \ldots \mid \mu x : U C . M
\]
Examples of fibred adjunction models

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

• given the EM-adjunction \( F^T \dashv U^T : \text{CPO}^T \rightarrow \text{CPO} \),
we can lift it to \( \widehat{F}_D \dashv \widehat{U}_D : \text{CFam}(\text{CPO}^T) \rightarrow \text{CFam}(\text{CPO}) \)
between
\[
\begin{align*}
\text{cfam}_{\text{CPO}} &: \text{CFam}(\text{CPO}) \rightarrow \text{CPO} \\
\text{cfam}_{\text{CPO}^T} &: \text{CFam}(\text{CPO}^T) \rightarrow \text{CPO}
\end{align*}
\]

• resulting in

\[
\begin{align*}
(\llbracket \Gamma \rrbracket, \llbracket A \rrbracket) & \in \text{CFam}(\text{CPO}) & (\llbracket \Gamma \rrbracket \in \text{CPO}, \llbracket A \rrbracket \in \llbracket \Gamma \rrbracket \rightarrow \text{CPO}^{EP}) \\
(\llbracket \Gamma \rrbracket, \llbracket C \rrbracket) & \in \text{CFam}(\text{CPO}^T) & (\llbracket C \rrbracket \in \llbracket \Gamma \rrbracket \rightarrow (\text{CPO}^T)^{EP})
\end{align*}
\]

• if \( T \) supports a least zero-ary op., then it also models recursion
\[
M ::= \ldots \mid \mu x : U C . M
\]
Examples of fibred adjunction models

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

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

  $\mathcal{V} \xrightarrow{T} \mathcal{V}$

  \[ p \downarrow \quad \downarrow p \]

  \[ \mathcal{B} \]

  and $p(\eta_A) = \text{id}_{p(A)}$; $p(\mu_A) = \text{id}_{p(A)}$

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

  $\mathcal{V} \xrightarrow{F^T} \mathcal{V}^T$

  \[ \perp \]

  \[ \mathcal{V} \]

  $\downarrow U^T$ \hspace{1cm} \[ \downarrow p^T \]

  \[ \mathcal{B} \]

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

- 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

\[
\Pi^T_A(B, \beta) \overset{\text{def}}{=} (\Pi_A(B), \beta_{\Pi^T_A})
\]

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

\[
\sigma_A : \Sigma_A \circ T \rightarrow T \circ \Sigma_A \quad (A \in \mathcal{V})
\]

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

\[
\Sigma^T_A(B, \beta) \overset{\text{def}}{=} (\Sigma_A(B), \beta_{\Sigma^T_A})
\]

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

\[
\Sigma^T_A(B, \beta) \overset{\text{def}}{=} F^T(\Sigma_A(B))_{/\equiv}
\]
Examples of fibred adjunction models

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

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

  \[ \Pi^T_A(B, \beta) \overset{\text{def}}{=} (\Pi_A(B), \beta_{\Pi^T_A}) \]

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

  \[ \sigma_A : \Sigma_A \circ T \longrightarrow T \circ \Sigma_A \quad (A \in V) \]

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

  \[ \Sigma^T_A(B, \beta) \overset{\text{def}}{=} (\Sigma_A(B), \beta_{\Sigma^T_A}) \]

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

  \[ \Sigma^T_A(B, \beta) \overset{\text{def}}{=} F^T(\Sigma_A(B)) \]


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

  $$\Pi^T_A(B, \beta) \overset{\text{def}}{=} (\Pi_A(B), \beta_{\Pi^T_A})$$

- **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

  $$\Sigma^T_A(B, \beta) \overset{\text{def}}{=} (\Sigma_A(B), \beta_{\Sigma^T_A})$$

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

  $$\Sigma^T_A(B, \beta) \overset{\text{def}}{=} \left. F^T(\Sigma_A(B)) \right/ \equiv$$
Algebraic effects
(operations and equations)
Algebraic effects – ops. and eqs.

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

- signatures of *dependently typed operation symbols*

$$\vdash I \quad x_i : I \vdash O \quad I \text{ and } O \text{ are pure value types}$$

$$\text{op} : (x_i : I) \rightarrow O$$

- equipped with *equations* on derivable effect terms

In eMLTT:

$$M ::= \ldots \mid \text{op}^C_V(x.M)$$

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

$$\Gamma \vdash V : I \quad \Gamma, x : O[V/x_i] \vdash M : C \quad \Gamma \mid z : C \vdash K : D$$

$$\Gamma \vdash K[\text{op}^C_V(x.M)/z] = \text{op}^D_V(x.K[M/z]) : D$$

Sound semantics: Based on families fibrations and Law. theories

- $p : \text{Fam}(\text{Set}) \rightarrow \text{Set}$ and $q : \text{Fam}(\text{Mod}(\mathcal{L}_{\mathcal{T}_{\text{eff}}}, \text{Set})) \rightarrow \text{Set}$
Algebraic effects – ops. and eqs.

**Fibred effect theories** $\mathcal{T}_{\text{eff}}$:  
- signatures of dependently typed operation symbols  
  
  \[
  \frac{\Gamma \vdash I \quad x_i : I \vdash O}{\text{and } O \text{ are pure value types}} \quad \text{op} : (x_i : I) \rightarrow O
  \]

- equipped with equations on derivable effect terms

**In eMLTT:**  
\[
M ::= \ldots \mid \text{op} \frac{C}{V}(x.M)
\]

**General algebraicity equations** (in addition to eff. th. eqs.):  
\[
\frac{\Gamma \vdash V : I \quad \Gamma, x : O[V/x_i] \vdash M : C \quad \Gamma \mid z : C \vdash K : D}{\Gamma \vdash K[\text{op} \frac{C}{V}(x.M)/z] = \text{op} \frac{D}{V}(x.K[M/z]) : D}
\]

**Sound semantics:** Based on families fibrations and Law. theories  
- $p : \text{Fam}(\text{Set}) \rightarrow \text{Set}$ and $q : \text{Fam}(\text{Mod}(\mathcal{L}_{\mathcal{T}_{\text{eff}}}, \text{Set})) \rightarrow \text{Set}$
Algebraic effects – ops. and eqs.

Fibred effect theories $T_{\text{eff}}$:
- signatures of dependently typed operation symbols
  \[
  \vdash I \quad x_i : I \vdash O \quad I \text{ and } O \text{ are pure value types} \]
  \[
  \text{op} : (x_i : I) \rightarrow O
  \]
- equipped with equations on derivable effect terms

In eMLTT:
\[
M ::= \ldots \mid \text{op}^C_V(x.M)
\]

General algebraicity equations (in addition to eff. th. eqs.):
\[
\Gamma \vdash V : I \quad \Gamma, x : O[V/x_i] \vdash \text{M} : C \quad \Gamma \mid z : C \vdash \text{K} : D
\]
\[
\Gamma \vdash \text{K}[\text{op}^C_V(x.M)/z] = \text{op}^D_V(x.K[M/z]) : D
\]

Sound semantics: Based on families fibrations and Law. theories
- $p : \text{Fam(Set)} \rightarrow \text{Set}$ and $q : \text{Fam(Mod}(\mathcal{L}_{T_{\text{eff}}}, \text{Set}) \rightarrow \text{Set}$
Algebraic effects – ops. and eqs.

Fibred effect theories $\mathcal{T}_{\text{eff}}$:
- signatures of dependently typed operation symbols
  \[
  \vdash I \quad x_i : I \vdash O \quad I \text{ and } O \text{ are pure value types} \quad \text{op} : (x_i : I) \rightarrow O
  \]
- equipped with equations on derivable effect terms

In eMLTT:
\[
M ::= \ldots \mid \text{op}^C_V(x.M)
\]

General algebraicity equations (in addition to eff. th. eqs.):
\[
\Gamma \vdash_V V : I \quad \Gamma, x : O[V/x_i] \vdash_c M : C \quad \Gamma \vdash_z C \vdash_h K : D \quad (\text{op} : (x_i : I) \rightarrow O)
\]
\[
\Gamma \vdash_c K[\text{op}^C_V(x.M)/z] = \text{op}^D_V(x.K[M/z]) : D
\]

Sound semantics: Based on families fibrations and Law. theories
- $p : \text{Fam}(\text{Set}) \rightarrow \text{Set}$ and $q : \text{Fam}(\text{Mod}(\mathcal{L}_{\mathcal{T}_{\text{eff}}}, \text{Set})) \rightarrow \text{Set}$
Algebraic effects – examples

Example 1 (interactive IO):

- read : 1 $\rightarrow$ Chr
  write : Chr $\rightarrow$ 1
- no equations

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

- $\Diamond \vdash \text{Loc} \\
  \ell : \text{Loc} \vdash \text{Val} \\
  \Diamond \vdash \text{isDec}_{\text{Loc}} : \Pi \ell : \text{Loc}. \Pi \ell' : \text{Loc}. (\ell =_{\text{Loc}} \ell') + (\ell =_{\text{Loc}} \ell' \rightarrow 0)$
- get : $(\ell : \text{Loc}) \rightarrow \text{Val} \\
  \text{put} : (\Sigma \ell : \text{Loc}. \text{Val}) \rightarrow 1$
- five equations (two of them branching on isDec_{\text{Loc}})

Example 3 (dep. typed update monads $T X \overset{\text{def}}{=} \Pi_{s : S} P s \times X$)
Algebraic effects – examples

Example 1 (interactive IO):
  • read : 1 ↦ Chr
  • write : Chr ↦ 1
  • no equations

Example 2 (global state with location-dependent store type):
  • □ ⊢ Loc
    • \ell : Loc ⊢ Val
    • □ ⊢ isDec_{Loc} : \Pi \ell : Loc . \Pi \ell' : Loc . (\ell \equiv_{Loc} \ell') + (\ell \equiv_{Loc} \ell' \rightarrow 0)
  • get : (\ell : Loc) ↦ Val
    • put : (\Sigma \ell : Loc . Val) ↦ 1
  • five equations (two of them branching on isDec_{Loc})

Example 3 (dep. typed update monads \( T X \overset{\text{def}}{=} \Pi_{s : S} . P s \times X \))
Algebraic effects – examples

Example 1 (interactive IO):
- read : 1 \rightarrow \text{Chr}
  \quad (\text{Chr} \overset{\text{def}}{=} 1 + \ldots + 1)
- write : \text{Chr} \rightarrow 1
- no equations

Example 2 (global state with location-dependent store type):
- \diamond \vdash \text{Loc}
  \quad \ell : \text{Loc} \vdash \text{Val}
- \diamond \vdash \text{isDec}_{\text{Loc}} : \Pi \ell : \text{Loc}. \Pi \ell' : \text{Loc}. (\ell =_{\text{Loc}} \ell') + (\ell =_{\text{Loc}} \ell' \rightarrow 0)
- get : (\ell : \text{Loc}) \rightarrow \text{Val}
  \quad \text{put} : (\Sigma \ell : \text{Loc} . \text{Val}) \rightarrow 1
- five equations (two of them branching on isDec_{\text{Loc}})

Example 3 (dep. typed update monads $T X \overset{\text{def}}{=} \prod_{s : S} P s \times X$)
Handlers of algebraic effects
(for programming and extrinsic reasoning)
Handlers of alg. effects – for programming

**Idea:** Generalisation of exception handlers \[\text{[Plotkin, Pretnar'09]}\]

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

Usual term-level presentation:

$$
\Gamma \vdash M \text{ handled with } \{ op_{x^v}(x_k) \mapsto N_{op} \}_{op \in \mathcal{T}_{\text{eff}}} \text{ to } y : A \text{ in } C \quad N_{\text{ret}} : \mathcal{C}
$$

satisfying

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

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

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., $S \to X \times S$)
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:**

$$\Gamma \vdash M \text{ handled with } \{ \text{op}_{x,v}(x_k) \mapsto N_{\text{op}} \}_{\text{op} \in \mathcal{T}_{\text{eff}}} \text{ to } y:A \text{ in } \mathcal{C} \ N_{\text{ret}} : \mathcal{C}$$

satisfying

- $(\text{return } V)$ handled with $\{ \ldots \}_{\text{op} \in \mathcal{T}_{\text{eff}}} \text{ to } y:A \text{ in } N_{\text{ret}} = N_{\text{ret}}[V/x]$,
- $(\text{op}_{V}(x.M))$ handled with $\{ \ldots \}_{\text{op} \in \mathcal{T}_{\text{eff}}} \text{ to } y:A \text{ in } N_{\text{ret}} = N_{\text{op}}[V/x_v][\ldots/x_k]$

**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., $S \to X \times S$)
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:

$$\Gamma \vdash M \text{ handled with } \{ op_{x,v}(x_k) \mapsto N_{op} \}_{op \in \mathcal{T}_{eff}} \text{ to } y : A \text{ in}_C N_{ret} : C$$

satisfying

\begin{itemize}
  \item \text{(return V) handled with } \{ \ldots \}_{op \in \mathcal{T}_{eff}} \text{ to } y : A \text{ in } N_{ret} = N_{ret}[V/x]
  \\
  \item \text{(op}_C V(x.M)) \text{ handled with } \{ \ldots \}_{op \in \mathcal{T}_{eff}} \text{ to } y : A \text{ in } N_{ret} = N_{op}[V/x_v][\ldots/x_k]
\end{itemize}

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., $S \to X \times S$)
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:

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

satisfying

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

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

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

\[
M \text{ handled with } \{ \text{op}_x(x_k) \mapsto V_{\text{op}} \}_{\text{op} \in T_{\text{eff}}} \text{ to } y:A \text{ in } B \text{ V}_{\text{ret}}
\]

we can define natural predicates (essentially, dependent types)

\[
\Gamma \vdash P : UFA \rightarrow U
\]

by

- equipping a universe \(U\) with an algebra for \(T_{\text{eff}}\) (sort of), 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 \(U\) closed under Nat, 1, 0, +, Σ, and Π
- a type-based treatment of handlers

\[
C ::= \ldots | \langle A; V_{\text{op}}; W_{\text{eq}} \rangle
\]
- function extensionality (actually, it's a bit more extensional)
Handlers of alg. effects – for reasoning

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

\[ M \text{ handled with } \{ \text{op}_x (x_k) \mapsto V_{\text{op}} \}_{\text{op} \in \mathcal{T}_{\text{eff}}} \text{ to } y : A \text{ in } B \text{ V}_{\text{ret}} \]

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 \( \mathcal{T}_{\text{eff}} \) (sort of), 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 Nat, 1, 0, +, \( \Sigma \), and \( \Pi \)

• a type-based treatment of handlers

\[ \mathcal{C} ::= \ldots | \langle A; \overrightarrow{V_{\text{op}}}; \overrightarrow{W_{\text{eq}}} \rangle \]

• function extensionality (actually, it's a bit more extensional)
Handlers of alg. effects – for reasoning

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

\[ M \text{ handled with } \{ \text{op}_x(y_k) \mapsto V_{\text{op}} \}_{\text{op} \in T_{\text{eff}}} \text{ to } y : A \text{ in } B \quad V_{\text{ret}} \]

we can define natural predicates (essentially, dependent types)

\[ \Gamma \vdash P : UFA \rightarrow U \]

by

- equipping a universe \( U \) with an algebra for \( T_{\text{eff}} \) (sort of), 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 \( U \) closed under Nat, 1, 0, +, \( \Sigma \), and \( \Pi \)
- a type-based treatment of handlers

\[ C ::= \ldots | (A; \overrightarrow{V}_{\text{op}}; \overrightarrow{W}_{\text{eq}}) \]
- 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 \( B \) \( V_{\text{ret}} \)

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 \( \mathcal{T}_{\text{eff}} \) (sort of), 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 eMLTTT with

- a universe \( \mathcal{U} \) closed under Nat, 1, 0, +, \( \Sigma \), and \( \Pi \)
- a type-based treatment of handlers \( \mathcal{C} ::= \ldots | \langle A; V_{\text{op}}; W_{\text{eq}} \rangle \)
- 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

\[
\diamond P \overset{\text{def}}{=} \lambda x : UFA. (\text{force } x) \text{ handled with } \{\ldots\}_{\text{op} \in \mathcal{T}_\text{IO}} \text{ to } y : A \text{ in } \mathcal{U}, P y
\]

using the handler given by

\[
V_{\text{read}} \overset{\text{def}}{=} \lambda x : (\Sigma x_v : 1. \text{Chr} \rightarrow \mathcal{U}). \overset{\sim}{\Sigma} y : \text{El}(\overset{\sim}{\text{Chr}}). (\text{snd } x) \, y
\]

\[
V_{\text{write}} \overset{\text{def}}{=} \lambda x : (\Sigma x_v : \text{Chr}. 1 \rightarrow \mathcal{U}). (\text{snd } x) \, \star
\]

- \( \diamond P \) corresponds to Evaluation Logic’s possibility modality

\[
\diamond P (\text{thunk } (\text{read}(x. \text{write} e (\text{return } V)))) = \overset{\sim}{\Sigma} x : \text{El}(\overset{\sim}{\text{Chr}}). P \, V
\]

- To get the necessity modality \( \square P \), just use \( \overset{\sim}{\Pi} x : \text{El}(\overset{\sim}{\text{Chr}}) \) in \( V_{\text{read}} \)
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

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

using the handler given by

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

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

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

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

- To get the necessity modality $\Box P$, just use $\hat{\Pi} x : \text{El(Chr)}$ 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

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

  using the handler given by

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

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

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

  $\Diamond P \left( \text{thunk} \left( \text{read}(x. \text{write}_{\mathcal{E}}'(\text{return } V)) \right) \right) = \Sigma x : \text{El}(\text{Chr}). P \ V$

  To get the necessity modality $\Box P$, just use $\hat{\Pi} x : \text{El}(\text{Chr})$ in $V_{\text{read}}$
 Handlers of alg. effects – for reasoning

Example 1 (Evaluation Logic style modalities):

- Given a predicate \( P : A \to U \) on return values, we define a predicate \( \Diamond P : UFA \to U \) on IO-computations as

\[
\Diamond P \overset{\text{def}}{=} \lambda x : UFA. (\text{force } x) \text{ handled with } \{ \ldots \}_{\text{op} \in T_{\text{IO}}} \text{ to } y : A \text{ in } \mathcal{U} \ P y
\]

using the handler given by

\[
V_{\text{read}} \overset{\text{def}}{=} \lambda x : (\Sigma x_v : 1 . \text{Chr} \to U). \tilde{\Sigma} y : \text{El}(\text{Chr}) . (\text{snd } x) \ y
\]

\[
V_{\text{write}} \overset{\text{def}}{=} \lambda x : (\Sigma x_v : \text{Chr} . 1 \to U) . (\text{snd } x) \star
\]

- \( \Diamond P \) corresponds to Evaluation Logic’s possibility modality

\[
\Diamond P \left( \text{thunk}(\text{read}(x . \text{write}_e'(\text{return } V))) \right) = \tilde{\Sigma} x : \text{El}(\text{Chr}) . P \ V
\]

- To get the necessity modality \( \Box P \), just use \( \tilde{\Pi} x : \text{El}(\text{Chr}) \) 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
  \[ Q : A \to S \to U \]
  \[ (S \overset{\text{def}}{=} \prod \ell : \text{Loc} \cdot \text{Val}) \]
  we define a precondition for stateful comps. on initial states
  \[ \text{wp}_Q : \text{UFA} \to S \to U \]
  by
  
  1) handling the given comp. into a state-passing function using
     \[ V_{\text{get}}, V_{\text{put}} \quad \text{on} \quad S \to (U \times S) \quad \text{and} \quad V_{\text{ret}} \quad \text{“=} \quad Q \]
  2) feeding in the initial state; and 3) projecting out \( U \)

- **Theorem:** \( \text{wp}_Q \) satisfies expected properties of WPs, e.g.,
  \[ \text{wp}_Q \left( \text{thunk} \left( \text{return} \ V \right) \right) = \lambda x_S : S . Q \ V \ x_S \]
  \[ \text{wp}_Q \left( \text{thunk} \left( \text{put}_{\langle \ell, V \rangle} (M) \right) \right) = \lambda x_S : S . \text{wp}_Q \left( \text{thunk} \ M \right) (x_S[\ell \mapsto V]) \]
Handlers of alg. effects – for reasoning

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

- Given a postcondition on return values and final states
  \[ Q : A \rightarrow S \rightarrow U \]
  \[ (S \overset{\text{def}}{=} \prod \ell : \text{Loc}.\text{Val}) \]
  we define a precondition for stateful comps. on initial states
  \[ \text{wp}_Q : UFA \rightarrow S \rightarrow U \]
  by
  
  1) handling the given comp. into a state-passing function using
  \[ V_{\text{get}}, V_{\text{put}} \text{ on } S \rightarrow (U \times S) \quad \text{and} \quad V_{\text{ret}} \overset{\text{“=} \text{”}}{=} Q \]
  2) feeding in the initial state; and
  3) projecting out \( U \)

- Theorem: \( \text{wp}_Q \) satisfies expected properties of WPs, e.g.,
  \[ \text{wp}_Q (\text{thunk} (\text{return } V)) = \lambda x_S : S. Q \lor x_S \]
  \[ \text{wp}_Q (\text{thunk} (\text{put}_{\langle \ell, V \rangle} (M))) = \lambda x_S : S. \text{wp}_Q (\text{thunk } M) (x_S[\ell \mapsto V]) \]
Handlers of alg. effects – for reasoning

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

- Given a postcondition on return values and final states

\[ Q : A \rightarrow S \rightarrow \mathcal{U} \]  \hspace{1cm} (S \overset{\text{def}}{=} \prod \ell : \text{Loc}.\text{Val})

we define a precondition for stateful comps. on initial states

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

by

1) handling the given comp. into a state-passing function using \[ V_{\text{get}}, V_{\text{put}} \] on \[ S \rightarrow (\mathcal{U} \times S) \] and \[ V_{\text{ret}} \text{ “=}” Q \]

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

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

\[ \text{wp}_Q \left( \text{thunk} \left( \text{return} \ V \right) \right) = \lambda x_S : S . \ Q \ V \ x_S \]

\[ \text{wp}_Q \left( \text{thunk} \left( \text{put}_{\ell,V}(M) \right) \right) = \lambda x_S : S . \text{wp}_Q \left( \text{thunk} \ M \right) \left( x_S[\ell \mapsto V] \right) \]
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 \( \land, \lor, \ldots \)

- We can define a rel. between comps. and protocols as follows:

\[ \text{Allowed} : \mathcal{UFA} \rightarrow \text{Protocol} \rightarrow \mathcal{U} \]

by handling the given computation using

\[ V_{\text{read}}, V_{\text{write}} \text{ on } \text{Protocol} \rightarrow \mathcal{U} \]

where

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

\[ V_{\text{write}} \langle V , V_{\text{wk}} \rangle (w \: P \: Pr') \text{ def } = \hat{\Sigma} x : \text{El}(P \: V) . V_{\text{wk}} \: \star \: Pr' \text{ def } = \hat{0} \]
 Handlers of alg. effects – for reasoning

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

- Assuming an inductive type of **IO-protocols**, given by
  
  \[
  \begin{align*}
  e &: \text{Protocol} \\
  r &: (\text{Chr} \rightarrow \text{Protocol}) \rightarrow \text{Protocol} \\
  w &: (\text{Chr} \rightarrow \mathcal{U}) \rightarrow \text{Protocol} \rightarrow \text{Protocol}
  \end{align*}
  \]

  and potentially also by \(\land, \lor, \ldots\)

- We can define a rel. between comps. and protocols as follows:

  \[
  \text{Allowed} : \mathcal{UFA} \rightarrow \text{Protocol} \rightarrow \mathcal{U}
  \]

  by handling the given computation using

  \[
  \begin{align*}
  V_{\text{read}}, V_{\text{write}} & \quad \text{on} \quad \text{Protocol} \rightarrow \mathcal{U} \\
  \end{align*}
  \]

  where

  \[
  \begin{align*}
  V_{\text{read}} \langle \_, V_{\text{rk}} \rangle (r \ Pr') & \quad \text{def} = \hat{\Pi} x : \text{El}(\text{Chr}). (V_{\text{rk}} x) (Pr' x) \\
  V_{\text{write}} \langle V, V_{\text{wk}} \rangle (w \ P \ Pr') & \quad \text{def} = \hat{\Sigma} x : \text{El}(P \ V). V_{\text{wk}} \star Pr'
  \end{align*}
  \]
Handlers of alg. effects – for reasoning

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

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

\[ e : \text{Protocol} \quad r : (\text{Chr} \to \text{Protocol}) \to \text{Protocol} \]
\[ w : (\text{Chr} \to \mathcal{U}) \to \text{Protocol} \to \text{Protocol} \]

and potentially also by \( \land, \lor, \ldots \)

• We can define a rel. between comps. and protocols as follows:

\[ \text{Allowed} : \mathcal{UFA} \to \text{Protocol} \to \mathcal{U} \]

by handling the given computation using

\[ V_{\text{read}}, V_{\text{write}} \quad \text{on} \quad \text{Protocol} \to \mathcal{U} \]

where

\[ V_{\text{read}} \langle - , V_{rk} \rangle (r \ Pr') \overset{\text{def}}{=} \hat{\Pi} x : \text{El(Chr)} . (V_{rk} x) (Pr' x) \]
\[ V_{\text{write}} \langle V, V_{wk} \rangle (w \ P \ Pr') \overset{\text{def}}{=} \hat{\Sigma} x : \text{El(P V)} . V_{wk} \star Pr' \overset{\text{def}}{=} \hat{0} \]
Conclusion

At a high-level, the presented work was about combining

dependent types and computational effects

In particular, you saw

• a clean core calculus 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

Some items of future work:

• uniform account of the various handler-defined predicates
• more expressive comp. types (par. adjunctions, Dijkstra monads)
Thank you!

D. Ahman. 

D. Ahman, N. Ghani, G. Plotkin. 
**Dependent Types and Fibred Computational Effects.** (FoSSaCS’16)

D. Ahman. 
**Handling Fibred Computational Effects.** (POPL’18)