A fibrational view on computational effects
(dependent types + computational effects)

Danel Ahman
Prosecco Team, Inria Paris

Edinburgh, 28 November 2017
Overview – 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

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

which in turn could be used to type a sorting function

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

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

The Curry-Howard correspondence:

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

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

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

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

which in turn could be used to type a sorting function

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

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

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

which in turn could be used to type a sorting function

\[ \forall \exists \forall \text{sort} : \prod \ell : \text{List Nat}. \sum \ell' : \text{List Nat}. \left( \text{Sorted}(\ell') \times \ldots \right) \]

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

Examples:

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

Meta-languages and models: based on

- monads ($\lambda_c$, $\lambda_{ML}$, FGCBV) (Moggi, Levy)
  
  \[ T : \mathcal{V} \rightarrow \mathcal{V} \]

- adjunctions (CBPV, EEC) (Levy, Egger et al.)
  
  \[ F : \mathcal{V} \rightarrow \mathcal{C} \quad U : \mathcal{C} \rightarrow \mathcal{V} \]

- algebraic presentations (Plotkin and Power)
  
  \[ \text{get} : 1 \rightarrow S \quad \text{put} : S \rightarrow 1 \quad + \text{equations} \]
Overview – putting the two together

We investigate the combination of

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

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

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 \(\text{(via a clean core language)}\)
• 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 \( (\Pi, \Sigma, V =_A W, \ldots) \)
- computational effects (state, nondeterminism, IO, ...)

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

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
- computational effects

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 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 Sorted(\(\ell\)) to 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 $\text{Sorted}^+(c)$, where $c : T(\text{List Chr})$
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
- 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 \texttt{Sorted}(\ell) to \texttt{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 $\Gamma \vdash C$ (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 $\lambda_{\text{ML}}$ 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 $\Gamma \vdash C$ (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 $\lambda_{ML}$ 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 \( \Gamma \vdash C \) (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 \( \lambda_{ML} \) 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

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

is not correct any more because \( x \) can appear free in the type \( C(x) \)

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

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

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

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

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

- then it would be natural to let $C$ depend on $x$, e.g.,
  \[ x : \text{Bool} \vdash C(x) \overset{\text{def}}{=} \text{if } x \text{ then “allow fread, fwrite, and fclose” } \]
  \[ \text{else “allow fopen”} \]

(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 $C$:

\[
\begin{align*}
\Gamma \vdash M : F A & \quad \Gamma \vdash C \quad \Gamma, x : A \vdash N : C \\
\Gamma \vdash M \text{ to } x : A \text{ in } N : C
\end{align*}
\]

**But:** Sometimes it is useful if $C$ 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 $C$ depend on $x$, e.g.,

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

(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

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

**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

\[
\begin{align*}
\Gamma \vdash M : T A \\
\Gamma, x : A \vdash N : T B(x)
\end{align*}
\]

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

**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!?!?

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

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

**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

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

**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 $C$ 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

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

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

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

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

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

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

**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

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

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

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

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

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

**Option 4:** 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 \( \text{return } 4, \text{return } 2 \)} \text{ 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 $C[4/x] + C[2/x]$ 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 $\mathcal{C}$ in seq. comp. (Option 1)

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

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

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

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

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

\[ \mathcal{C}[4/x] + \mathcal{C}[2/x] \triangleq F \left( U (\mathcal{C}[4/x]) + U (\mathcal{C}[2/x]) \right) \equiv \]

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:** \( \text{MLTT} + \text{thunks} + \ldots \)

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

- \( U_C \) 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 types: $\text{MLTT} + \text{thunks} + \ldots$

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

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

Computation types: dependent-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 $A \rightarrow C$ 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. terms

Value terms: MLTT + thunks + ...

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

- equational theory based on intensional MLTT

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

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

\[ M, N ::= \text{force} V \]
\[ \mid \text{return} V \]
\[ \mid M \text{ to } x:A \text{ in } N \]
\[ \mid \lambda x:A. M \]
\[ \mid MV \]
\[ \mid \langle V, M \rangle \quad \text{(comp. } \Sigma \text{ intro.)} \]
\[ \mid M \text{ to } \langle x:A, z:C \rangle \text{ in } K \quad \text{(comp. } \Sigma \text{ elim.)} \]

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

\[ M, N ::= \text{force } V \]
\[ \mid \text{return } V \]
\[ \mid M \text{ to } x : A \text{ in } N \]
\[ \mid \lambda x : A . M \]
\[ \mid MV \]
\[ \mid \langle V, M \rangle \] \hspace{1cm} (comp. Σ intro.)
\[ \mid M \text{ to } \langle x : A, z : C \rangle \text{ in } K \] \hspace{1cm} (comp. Σ elim.)

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

\[
\begin{align*}
K, L & ::= z & \text{(linear comp. vars.)} \\
       & | K \text{ to } x : A \text{ in } M \\
       & | \lambda x : A. K \\
       & | KV \\
       & | \langle V, K \rangle & \text{(comp. } \Sigma \text{ intro.)} \\
       & | K \text{ to } \langle x : A, z : C \rangle \text{ in } L & \text{(comp. } \Sigma \text{ elim.)}
\end{align*}
\]

**Typing judgments:**

- $\Gamma \vdash V : A$
- $\Gamma \vdash M : C$
- $\Gamma \vdash z : C \vdash K : D$ \hspace{1cm} (linear in $z$; comp. bound to $z$ happens first)
Note: We need to define $K$ in such a way that the intended left-to-right evaluation order is preserved, e.g., consider

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

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

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

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

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

$$| KV$$

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

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

Typing judgments:

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

\[ \Gamma \vdash c : M \quad \Gamma, x : A \vdash N : C(x) \]

\[ \Gamma \vdash \Sigma x : A \cdot C(x) \]

\[ \Gamma \vdash M \to x : A \in \langle x, N \rangle : \Sigma x : A \cdot C(x) \]

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

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

\[ \Gamma \vdash U \left( \Sigma x : A \cdot FB(x) \right) \cong UF \left( \Sigma x : A \cdot B(x) \right) = T \left( \Sigma x : A \cdot B(x) \right) \]
Categorical semantics of eMLTT
(fibrations + adjunctions)
Fibred adjunction models – value part

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

\[
\begin{array}{c}
\forall \\
p \\
\downarrow \\
\downarrow \\
\B \\
\end{array}
\]

allowing us to define a partial interpretation fun. \([\phantom{-}\phantom{-}]\), that maps:

- a context \( \Gamma \) to and object \([\Gamma]\) in \( \mathcal{B} \), with
  - \([\Diamond] \overset{\text{def}}{=} 1\)
  - \([\Gamma, x : A] \overset{\text{def}}{=} \{[\Gamma; A]\}\) \hspace{1cm} (if \( x \not\in \text{Vars}(\Gamma) \) and \([\Gamma; A]\) is defined)
- a context \( \Gamma \) and a value type \( A \) to an object \([\Gamma; A]\) in \( \mathcal{V}_{[\Gamma]} \)
- a context \( \Gamma \) and a value term \( V \) to \([\Gamma; V] : 1_{[\Gamma]} \to A \) in \( \mathcal{V}_{[\Gamma]} \)
Fibred adjunction models – value part

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

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

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 $F \vdash U$, 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 $\llbracket \Gamma; C \rrbracket$ in $C_{[\Gamma]}$
- a ctx. $\Gamma$ and a comp. term $M$ to $\llbracket \Gamma; M \rrbracket : 1_{[\Gamma]} \rightarrow U(C)$ in $V_{[\Gamma]}$
- a ctx. $\Gamma$, a comp. var. $z$, a comp. type $\underline{C}$, and a hom. term $K$ to $\llbracket \Gamma; z : C; K \rrbracket : \llbracket \Gamma; C \rrbracket \rightarrow D$ in $C_{[\Gamma]}$
Fibred adjunction models – effects part

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

\[
\begin{array}{ccc}
V & \xleftarrow{q} & \mathcal{B} \\
\downarrow & & \downarrow \beta \\
\downarrow & \leftarrow \{ - \} & \rightarrow \{ - \}
\end{array}
\]

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 split fibred pre-enrichment in \( p \) (hom. function type \( \rightarrow \))
Fibred adjunction models – correctness

**Theorem (Soundness):**
- If $\Gamma \vdash C$, then $[\Gamma; C] \in \mathcal{C}_{[\Gamma]}$
- If $\Gamma \vdash \mathsf{M} : C$, then $[\Gamma; \mathsf{M}] : 1_{[\Gamma]} \rightarrow U([\Gamma; C])$
- If $\Gamma \vdash z : C \vdash K : D$, 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

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

\[
\begin{array}{ccc}
\mathcal{V} & \xRightarrow{1} & \mathcal{B} \\
\end{array}
\]

\[
\begin{array}{ccc}
\mathcal{B} & \xRightarrow{F} & \mathcal{V} \\
\end{array}
\]

\[
\begin{array}{ccc}
\mathcal{B} & \xRightarrow{U} & \mathcal{C} \\
\end{array}
\]
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 \([A] \in \Gamma \to \text{Set}\))
- \((\Gamma, C) \in \text{Fam}(\mathcal{D})\) (where \([C] \in \Gamma \to \mathcal{D}\))

**Example 4** (continuous families and CPO-enriched monads):
- \((\Gamma, A) \in \text{CFam}(\text{CPO})\) (where \([A] \in \Gamma \to \text{CPO}^{EP}\))
- \((\Gamma, C) \in \text{CFam}(\text{CPO}^T)\) (where \([C] \in \Gamma \to (\text{CPO}^T)^{EP}\))

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

Example 4 (continuous families and CPO-enriched monads):
- \(([[\Gamma], [A]] \in \text{CFam}(\text{CPO})) \quad (\text{where } [A] \in [\Gamma] \rightarrow \text{CPO}^{EP})\)
- \(([[\Gamma], [C]] \in \text{CFam}(\text{CPO}^T)) \quad (\text{where } [C] \in [\Gamma] \rightarrow (\text{CPO}^T)^{EP})\)

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*):
- \((\lceil \Gamma \rceil, \lceil A \rceil) \in \text{Fam}(\text{Set})\)  (where \(\lceil A \rceil \in \lceil \Gamma \rceil \to \text{Set}\))
- \((\lceil \Gamma \rceil, \lceil C \rceil) \in \text{Fam}(\mathcal{D})\)  (where \(\lceil C \rceil \in \lceil \Gamma \rceil \to \mathcal{D}\))

**Example 4** (*continuous families and CPO-enriched monads*):
- \((\lceil \Gamma \rceil, \lceil A \rceil) \in \text{CFam}(\text{CPO})\)  (where \(\lceil A \rceil \in \lceil \Gamma \rceil \to \text{CPO}^{\text{EP}}\))
- \((\lceil \Gamma \rceil, \lceil C \rceil) \in \text{CFam}(\text{CPO}^T)\)  (where \(\lceil C \rceil \in \lceil \Gamma \rceil \to (\text{CPO}^T)^{\text{EP}}\))

**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):
- $(\lfloor \Gamma \rfloor, \lfloor A \rfloor) \in \text{Fam}(\text{Set})$ (where $\lfloor A \rfloor \in \lfloor \Gamma \rfloor \rightarrow \text{Set}$)
- $(\lfloor \Gamma \rfloor, \lfloor C \rfloor) \in \text{Fam}(\mathcal{D})$ (where $\lfloor C \rfloor \in \lfloor \Gamma \rfloor \rightarrow \mathcal{D}$)

Example 4 (continuous families and CPO-enriched monads):
- $(\lfloor \Gamma \rfloor, \lfloor A \rfloor) \in \text{CFam}(\text{CPO})$ (where $\lfloor A \rfloor \in \lfloor \Gamma \rfloor \rightarrow \text{CPO}^{EP}$)
- $(\lfloor \Gamma \rfloor, \lfloor C \rfloor) \in \text{CFam}(\text{CPO}^T)$ (where $\lfloor C \rfloor \in \lfloor \Gamma \rfloor \rightarrow (\text{CPO}^T)^{EP}$)

Theorem: $\text{cod}_{\text{CPO}}$ is not suitable because CPO is not an LCCC.
Examples of fibred adjunction models

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

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

  \[
  \begin{array}{c}
  V \\
  \downarrow^p \\
  B
  \end{array}
  \xrightarrow{T}
  \begin{array}{c}
  V \\
  \downarrow^p \\
  B
  \end{array}
  \]

- we consider models based on the EM-resolution of $T$

  \[
  \begin{array}{c}
  V \\
  \downarrow^p \\
  B
  \end{array}
  \xleftarrow{F^T}
  \begin{array}{c}
  V^T \\
  \downarrow^{p^T} \\
  B
  \end{array}
  \xrightarrow{U^T}
  \begin{array}{c}
  V \\
  \downarrow^p \\
  B
  \end{array}
  \]

  where $\left( A \in V , \alpha : T(A) \rightarrow A \right) \in V^T$

- 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

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

• Proposition: 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

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

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

- **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

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

\[
F^T(\Sigma_A(B))_\equiv
\]

\[
F^T(\Sigma_A(B))_\equiv
\]
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

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

• **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

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

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

- signatures of dependently typed operation symbols

\[
\text{\cdot} \vdash I \quad x_i : I \vdash O \quad I \text{ and } O \text{ are pure value types}
\]

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

- equipped with equations on derivable effect terms

In eMLTT:

\[
M ::= \ldots | \text{op}_V^C(x.M)
\]

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

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

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

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

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

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

- signatures of dependently typed operation symbols
  \[
  \frac{\vdash I}{x_i : I \vdash O} \quad I \text{ and } O \text{ are pure value types} \quad \text{op} : (x_i : I) \rightarrow O
  \]
- 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.):

\[
\frac{\Gamma \vdash V : I \quad \Gamma, x : O[V/x_i] \vdash M : C \quad \Gamma \mid z : C \vdash K : D}{\Gamma \vdash K[\text{op}^C_V(x.M)/z] = \text{op}^D_V(x.K[M/z]) : D}
\]

Sound semantics: Based on families fibrations and Law. theories

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

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

- signatures of dependently typed operation symbols
  \[
  \begin{array}{c}
  \Gamma \vdash I \\
  x_i : I \\n  \Gamma \vdash O \\
  \end{array}
  \quad \text{I and O are pure value types}
  \quad \op : (x_i : I) \rightarrow O
  \]

- equipped with equations on derivable effect terms

In eMLTT:

\[
M ::= \ldots \mid \opC{V}(x.M)
\]

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

\[
\begin{array}{c}
\Gamma \vdash V : I \\
\Gamma, x : O[V/x_i] \vdash C \\
\Gamma \mid z : C \vdash D \\
\end{array}
\quad \Gamma \vdash \opD{V}(x.M/z) = \opC{V}(x.K[M/z]) : D
\]

Sound semantics: Based on families fibrations and Law. theories

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

Fibred effect theories $\mathcal{T}_{\text{eff}}$:
- signatures of dependently typed operation symbols

\[
\vdash I \quad x_i : I \vdash O \quad I \text{ and } O \text{ are pure value types} \\
\text{op : } (x_i : I) \rightarrow O
\]

- 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.):

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

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

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 – examples

Example 1 (interactive IO):

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

(\text{Chr} \overset{\text{def}}{=} 1 + \ldots + 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}} : \Pi \ell : \text{Loc} \cdot \Pi \ell' : \text{Loc}. (\ell =_{\text{Loc}} \ell') + (\ell =_{\text{Loc}} \ell' \rightarrow 0)
- get : (\ell : \text{Loc}) \rightarrow \text{Val}
- put : (\Sigma \ell : \text{Loc} \cdot \text{Val}) \rightarrow 1
- five equations (two of them branching on \text{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
  - ℓ : Loc ⊨ Val
  - ◇ ⊨ isDec_{Loc} : Π ℓ : Loc . Π ℓ' : Loc . (ℓ =_{Loc} ℓ') + (ℓ =_{Loc} ℓ' → 0)
- get : (ℓ : Loc) ↩ Val
- put : (Σ ℓ : Loc . Val) ↩ 1
- five equations (two of them branching on isDec_{Loc})

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

Example 1 (interactive IO):

- read : 1 → Chr
  write : Chr → 1

- no equations

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

- \[ \Diamond \vdash \text{Loc} \]
  \[ \ell : \text{Loc} \vdash \text{Val} \]
  \[ \Diamond \vdash \text{isDec}_{\text{Loc}} : \Pi \ell : \text{Loc} . \Pi \ell' : \text{Loc} . (\ell =_{\text{Loc}} \ell') + (\ell =_{\text{Loc}} \ell' \to 0) \]

- get : (\ell : \text{Loc}) → Val
  put : (\sum \ell : \text{Loc} . \text{Val}) → 1

- five equations (two of them branching on isDec_{\text{Loc}})

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

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

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 programming**

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

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

**Usual term-level presentation:**

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

satisfying

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

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

**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:

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

satisfying

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

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

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 programming**

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

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

**Usual term-level presentation:**

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

satisfying

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

**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

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

we can define natural predicates (essentially, dependent types)

\[ \Gamma \vdash P : \mathcal{U}_{\text{FA}} \rightarrow \mathcal{U} \]

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 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**

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

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

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

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

by

- equipping a **universe** \( U \) with an **algebra** for \( T_{\text{eff}} \), 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 \( U \) closed under Nat, 1, 0, +, Σ, and Π
- a **type-based treatment of handlers** \( C ::= \ldots | \langle A; V_{op}; W_{eq} \rangle \)
- function extensionality (actually, it's a bit more extensional)
Handlers of alg. effects – for reasoning

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

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

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 \( T_{eff} \), 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 Nat, 1, 0, +, \( \Sigma \), and \( \Pi \)
- a type-based treatment of handlers
- function extensionality (actually, it's a bit more extensional)
Handlers of alg. effects – for reasoning

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

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

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 \( \mathcal{T}_{\text{eff}} \), 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 \( C ::= \ldots \mid \langle A ; \overrightarrow{V_{\text{op}}} ; \overrightarrow{W_{\text{eq}}} \rangle \)
- 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

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

using the handler given by

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

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

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

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

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

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

using the handler given by

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

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

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

\[
\diamond P \ (\text{thunk}(\text{read}(x. \text{write}_{\epsilon'}(\text{return } V)))) = \hat{\Sigma} x: \text{El}(\hat{\text{Chr}}). P \ V
\]

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

Example 1 (Evaluation Logic style modalities):

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

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

  using the handler given by

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

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

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

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

- To get the necessity modality $\Box P$, we use $\Pi x : \text{El}(\hat{\text{Chr}})$ in $V_{read}$
**Example 1** (Evaluation Logic style modalities):

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

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

  using the handler given by

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

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

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

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

  - To get the necessity modality $\Box P$, we use $\hat{\Pi} x : \text{El} (\hat{\text{Chr}})$ in $V_{\text{read}}$. 
Example 2 (Dijkstra’s weakest precondition semantics for state):

- Given a postcondition on return values and final states
  \[ Q : A \to S \to U \]
  \[(S \overset{\text{def}}{=} \prod \ell : \text{Loc} . \text{Val}) \]
  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 \( V_{\text{get}}, V_{\text{put}} \) on \( S \to (U \times S) \) and \( V_{\text{ret}} \) “=” \( Q \)
  2) feeding in the initial state; and 3) projecting out \( U \)

- **Theorem:** \( \text{wp}_Q \) satisfies expected properties of WPs, e.g.,
  \[ \text{wp}_Q \left( \text{thunk} \left( \text{return} \ V \right) \right) = \lambda x_S : S . Q \ V x_S \]
  \[ \text{wp}_Q \left( \text{thunk} \left( \text{put}_{\ell, V} (M) \right) \right) = \lambda x_S : S . \text{wp}_Q \left( \text{thunk} \ M \right) (x_S[\ell \mapsto V]) \]
Example 2 (Dijkstra’s weakest precondition semantics for state):

- Given a postcondition on return values and final states

  \[ Q : A \rightarrow S \rightarrow \mathcal{U} \quad (S \overset{\text{def}}{=} \Pi \ell : \text{Loc} \cdot \text{Val}) \]

  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

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

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

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

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

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

\[ V_{\text{get}}, V_{\text{put}} \text{ on } S \rightarrow (U \times S) \text{ and } V_{\text{ret}} \overset{\text{“=”}}{=} Q \]

2) feeding in the **initial state**; and 3) projecting out \( U \)

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

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

\[
\text{wp}_Q (\text{thunk} (\text{put}_{\langle \ell, V \rangle} (M))) = \lambda x_S : S. \text{wp}_Q (\text{thunk } M) (x_S[\ell \mapsto V])
\]
**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 relation between comps. and protocols as follows:
  \[ \text{Allowed} : \mathcal{UFA} \rightarrow \text{Protocol} \rightarrow \mathcal{U} \]
  by handling the given computation using
  \[ V_{\text{read}}, V_{\text{write}} \quad \text{on} \quad \text{Protocol} \rightarrow \mathcal{U} \]
  where
  \[ V_{\text{read}} \langle _, V_{\text{rk}} \rangle (r \text{ Pr'}) \overset{\text{def}}{=} \hat{\Pi} x : \text{El(Chr)}. (V_{\text{rk}} x)(\text{Pr'} x) \]
  \[ V_{\text{write}} \langle V, V_{\text{wk}} \rangle (w \text{ P Pr'}) \overset{\text{def}}{=} \hat{\Sigma} x : \text{El(P V)}. V_{\text{wk}} * \text{Pr'} \overset{\text{def}}{=} 0 \]
 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 \(\&\), \(\lor\), \ldots\)

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

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

  by handling the given computation using

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

  where

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

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

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

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

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

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

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

  by handling the given computation using

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

  where

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

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

In particular, you saw
- a clean core 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.

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

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