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

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

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

which in turn could be used for typing sorting functions

$$\texttt{sort} : \overset{\forall}{\Pi}\, \ell : (\mathsf{List\ Nat})\,.\, \overset{\exists}{\Sigma}\, \ell' : (\mathsf{List\ Nat})\,.\, \Big(\mathsf{Sorted}(\ell') \overset{\wedge}{\times} \ldots\Big)$$

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

Background – dependent types

The Curry-Howard correspondence:

Simple Types	$\sim$	Propositional Logic	$(\mathsf{Nat}, \mathsf{String}, \dots)$
Dependent Types	$\sim$	Predicate Logic	$(\Sigma, \Pi, =, \dots)$

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

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

which in turn could be used for typing sorting functions

$$\mathtt{sort} : \overset{\forall}{\Pi}\, \ell : (\mathsf{List\ Nat}) \,.\, \overset{\exists}{\Sigma}\, \ell' : (\mathsf{List\ Nat}) \,.\, \Big(\mathsf{Sorted}(\ell') \overset{\wedge}{\times} \dots \Big)$$

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

Background – dependent types

The Curry-Howard correspondence:

Simple Types	$\sim$	Propositional Logic	$(\mathsf{Nat}, \mathsf{String}, \ldots)$
Dependent Types	$\sim$	Predicate Logic	$(\Sigma, \Pi, =, \ldots)$

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

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

which in turn could be used for typing sorting functions

$$\mathtt{sort} : \overset{\forall}{\Pi}\, \ell : (\mathsf{List\ Nat})\,.\,\overset{\exists}{\Sigma}\, \ell' : (\mathsf{List\ Nat})\,.\,\Big(\mathsf{Sorted}(\ell') \overset{\wedge}{\times} \ldots\Big)$$

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

Background – computational effects

Examples:

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

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

- monads (λ_c, λ_{ML}, FGCBV) (Moggi, Levy)

$$[\![\Gamma \vdash M : A]\!]_{\lambda_c} : [\![\Gamma]\!] \longrightarrow T[\![A]\!]$$

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

$$[\![\Gamma \vdash_v V : A]\!]_{CBPV} : [\![\Gamma]\!] \longrightarrow [\![A]\!] \qquad [\![\Gamma \vdash_c M : \underline{C}]\!]_{CBPV} : [\![\Gamma]\!] \longrightarrow U([\![\underline{C}]\!])$$

- algebraic presentations (Plotkin and Power)

$$\mathsf{get} : 1 \rightharpoonup S \qquad \mathsf{put} : S \rightharpoonup 1 \qquad (+ \text{ equations})$$

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

Two guiding problems

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

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

Two guiding problems

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

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
- discover smth. interesting

Two guiding problems

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

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

Two guiding problems

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

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 (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 $\mathsf{Sorted}(\ell)$ to $\mathsf{Sorted}^{\dagger}(c)$, where $c : T(\mathsf{List\ Chr})$

$$\mathsf{Sorted}^{\dagger}(\texttt{receive}(y.\,\texttt{return}\,y)) = \langle\texttt{receive}\rangle(y.\,\mathsf{Sorted}(y))$$

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

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(ℓ) to Sorted†(c), where $c : T(\text{List Chr})$

$$\text{Sorted}^\dagger(\texttt{receive}(y.\,\texttt{return}\, y)) = \langle\texttt{receive}\rangle(y.\,\text{Sorted}(y))$$

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

Effectful programs in types

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

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 $\mathrm{Sorted}(\ell)$ to $\mathrm{Sorted}^{\dagger}(c)$, where $c : T(\mathrm{List\ Chr})$

$$\mathrm{Sorted}^{\dagger}(\mathtt{receive}(y.\,\mathtt{return}\,y)) = \langle\mathtt{receive}\rangle(y.\,\mathrm{Sorted}(y))$$

  - for this Sorted needs to be a T-algebra
- (cf. recent papers by Pédrot and Tabareau; and 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 $\Gamma \vdash \underline{C}$ (dep. typed CBPV/EEC)
- where Γ contains only value variables $x_1 : A_1, \ldots, x_n : A_n$

Could have also considered Moggi's λ_{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 $\Gamma \vdash \underline{C}$ (dep. typed CBPV/EEC)
- where Γ contains only value variables $x_1 : A_1, \ldots, x_n : A_n$

Could have also considered Moggi's λ_{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 $\Gamma \vdash \underline{C}$ (dep. typed CBPV/EEC)
- where Γ contains only value variables $x_1 : A_1, \ldots, x_n : A_n$

Could have also considered Moggi's λ_{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

$$\frac{\Gamma \vDash_{\mathsf{c}} M : F\,A \qquad \Gamma, x{:}A \vDash_{\mathsf{c}} N : \underline{C}(x)}{\Gamma \vDash_{\mathsf{c}} M \;\texttt{to}\; x{:}A \;\texttt{in}\; N : \underline{C}(x)}$$

is not correct any more because potentially

$$x \in FV(\underline{C})$$

in the conclusion

Assigning types to effectful programs

Aim: To fix the typing rule of sequential composition

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

$$\frac{\Gamma \vdash_{\mathsf{c}} M : F\,A \qquad \Gamma \vdash \underline{C} \qquad \Gamma, x\!:\!A \vdash_{\mathsf{c}} N : \underline{C}}{\Gamma \vdash_{\mathsf{c}} M \ \mathtt{to}\ x\!:\!A \ \mathtt{in}\ N : \underline{C}}$$

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

- say we consider

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

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

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

(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 $\underline{C}$: [Levy'04]

$$\frac{\Gamma \vDash_{c} M : F\,A \qquad \Gamma \vdash \underline{C} \qquad \Gamma, x\!:\!A \vDash_{c} N : \underline{C}}{\Gamma \vDash_{c} M \texttt{ to } x\!:\!A \texttt{ in } N : \underline{C}}$$

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

- say we consider

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

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

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

(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 $\underline{C}$: [Levy'04]

$$\frac{\Gamma \vDash_{\mathsf{c}} M : F\,A \qquad \Gamma \vdash \underline{C} \qquad \Gamma, x\!:\!A \vDash_{\mathsf{c}} N : \underline{C}}{\Gamma \vDash_{\mathsf{c}} M \mathtt{~to~} x\!:\!A \mathtt{~in~} N : \underline{C}}$$

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

- say we consider

$$\mathtt{fopen}\,\big(\mathtt{return~true}, \mathtt{return~false}\big) \mathtt{~to~} x\!:\!\mathsf{Bool} \mathtt{~in~} N$$

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

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

(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

$$\Gamma \vdash_c M \texttt{ to } x\!:\!A \texttt{ in } N : M \texttt{ to } x\!:\!A \texttt{ in } \underline{C}$$

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

Option 3: In the monadic metalanguage λ_{ML}, one could try

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

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

$$\Gamma \vDash_{c} M \ \texttt{to}\ x\!:\!A\ \texttt{in}\ N : M\ \texttt{to}\ x\!:\!A\ \texttt{in}\ \underline{C}$$

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

Option 3: In the monadic metalanguage λ_{ML}, one could try

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

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

$$\Gamma \vDash_c M \;\texttt{to}\; x\!:\!A \;\texttt{in}\; N : M \;\texttt{to}\; x\!:\!A \;\texttt{in}\; \underline{C}$$

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

Option 3: In the monadic metalanguage λ_{ML}, one could try

$$\frac{\Gamma \vdash M : T\,A \qquad \Gamma, x\!:\!A \vdash N : T\,B(x)}{\Gamma \vdash M \;\texttt{to}\; x\!:\!A \;\texttt{in}\; N : T\,(\Sigma x : A.B)}$$

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

Our solution: We draw inspiration from algebraic effects

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

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

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

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

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

Idea: M denotes an element of the coproduct of algebras

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

which we generalise to A-indexed coproducts, i.e., a comp. Σ-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 $\underline{C}$ in seq. comp. (**Option 1**)

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

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

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

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

Idea: M denotes an element of the coproduct of algebras

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

which we generalise to A-indexed coproducts, i.e., a comp. Σ-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 $\underline{C}$ in seq. comp. (**Option 1**)

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

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

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

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

Idea: M denotes an element of the coproduct of algebras

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

which we generalise to A-indexed coproducts, i.e., a comp. Σ-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 $\underline{C}$ in seq. comp. (**Option 1**)

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

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

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

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

Idea: M denotes an element of the coproduct of algebras

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

which we generalise to A-indexed coproducts, i.e., a comp. Σ-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 $\underline{C}$ in seq. comp. (**Option 1**)

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

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

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

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

Idea: M denotes an element of the coproduct of algebras

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

which we generalise to A-indexed coproducts, i.e., a comp. Σ-type

Putting these ideas together

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

eMLTT – value and comp. types

Value types: MLTT + thunks + ...

$$A, B ::= \mathsf{Nat} \mid 1 \mid 0 \mid \Pi\, x : A \,.\, B \mid \Sigma\, x : A \,.\, B \mid V =_A W \mid U\underline{C} \mid \ldots$$

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

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

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

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

eMLTT – value and comp. types

Value types: MLTT + thunks + ...

$$A, B ::= \mathsf{Nat} \mid 1 \mid 0 \mid \Pi\, x : A \,.\, B \mid \Sigma\, x : A \,.\, B \mid V =_A W \mid U\underline{C} \mid \ldots$$

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

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

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

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

eMLTT – value and comp. terms

Value terms: MLTT + thunks + ...

$$V, W \ ::= \ x \ | \ \mathtt{zero} \ | \ \mathtt{succ}\, V \ | \ \ldots \ | \ \mathtt{thunk}\ M \ | \ \ldots$$

- equational theory based on intensional MLTT

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

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

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

eMLTT – value and comp. terms

Value terms: MLTT + thunks + ...

$$V, W ::= x \mid \texttt{zero} \mid \texttt{succ}\, V \mid \ldots \mid \texttt{thunk}\ M \mid \ldots$$

- equational theory based on intensional MLTT

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

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

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

eMLTT – value and comp. terms

Value terms: MLTT + thunks + ...

$$V, W ::= x \mid \texttt{zero} \mid \texttt{succ}\, V \mid \ldots \mid \texttt{thunk}\ M \mid \ldots$$

- equational theory based on intensional MLTT

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

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

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

eMLTT – homomorphism terms

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

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

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

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

Typing judgments:

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

eMLTT – homomorphism terms

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

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

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

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

Typing judgments:

- $\Gamma \vdash_{\mathsf{v}} V : A$
- $\Gamma \vdash_{\mathsf{c}} M : \underline{C}$
- $\Gamma \,|\, z : \underline{C} \vdash_{\mathsf{h}} K : \underline{D}$ (linear in z; comp. bound to z happens first)

eMLTT – typing sequential composition

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

$$\frac{\Gamma \vDash_{\mathsf{c}} M : F\,A \qquad \Gamma \vdash \Sigma\, x\!:\!A\,.\,\underline{C}(x) \qquad \dfrac{\Gamma, x\!:\!A \vDash_{\mathsf{c}} N : \underline{C}(x)}{\Gamma, x\!:\!A \vDash_{\mathsf{c}} \langle x, N\rangle : \Sigma\, x\!:\!A\,.\,\underline{C}(x)}}{\Gamma \vDash_{\mathsf{c}} M \;\mathtt{to}\; x\!:\!A \;\mathtt{in}\; \langle x, N\rangle : \Sigma\, x\!:\!A\,.\,\underline{C}(x)}$$

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

$$\frac{\Gamma \vDash_{\mathsf{v}} \varepsilon_1 : \mathsf{Effect} \qquad \Gamma \vdash A \qquad \Gamma \vDash_{\mathsf{v}} \varepsilon_2 : A \to \mathsf{Effect}}{\Gamma \vdash T\,\varepsilon_1\,A\,\varepsilon_2}$$

in terms of standard parameterised effect-typing as

$$T\,\varepsilon_1\,A\,\varepsilon_2 \stackrel{\text{def}}{=} U_{\varepsilon_1}(\Sigma\, x\!:\!A\,.\,F_{\varepsilon_2\,x}\,1)$$

and thus naturally accommodate examples like

$$\mathtt{fopen}\,(\mathtt{return\ true}, \mathtt{return\ false})\;\mathtt{to}\;x\!:\!\mathsf{Bool}\;\mathtt{in}\;N$$

eMLTT – typing sequential composition

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

$$\frac{\Gamma \vdash_{\mathsf{c}} M : F\,A \qquad \Gamma \vdash \Sigma\, x\!:\!A\,.\,\underline{C}(x) \qquad \dfrac{\Gamma, x\!:\!A \vdash_{\mathsf{c}} N : \underline{C}(x)}{\Gamma, x\!:\!A \vdash_{\mathsf{c}} \langle x, N\rangle : \Sigma\, x\!:\!A\,.\,\underline{C}(x)}}{\Gamma \vdash_{\mathsf{c}} M \;\texttt{to}\; x\!:\!A \;\texttt{in}\; \langle x, N\rangle : \Sigma\, x\!:\!A\,.\,\underline{C}(x)}$$

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

$$\frac{\Gamma \vdash_{\mathsf{v}} \varepsilon_1 : \mathsf{Effect} \quad \Gamma \vdash A \quad \Gamma \vdash_{\mathsf{v}} \varepsilon_2 : A \to \mathsf{Effect}}{\Gamma \vdash T\,\varepsilon_1\,A\,\varepsilon_2}$$

in terms of standard parameterised effect-typing as

$$T\,\varepsilon_1\,A\,\varepsilon_2 \stackrel{\text{def}}{=} U_{\varepsilon_1}(\Sigma\, x\!:\!A\,.\,F_{\varepsilon_2\,x}\,1)$$

and thus naturally accommodate examples like

$$\texttt{fopen}\,(\texttt{return true}, \texttt{return false})\;\texttt{to}\; x\!:\!\mathsf{Bool}\;\texttt{in}\; N$$

eMLTT – typing sequential composition

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

$$\frac{\Gamma \vDash_{c} M : F\,A \qquad \Gamma \vdash \Sigma\, x{:}A\,.\,\underline{C}(x) \qquad \dfrac{\Gamma, x{:}A \vDash_{c} N : \underline{C}(x)}{\Gamma, x{:}A \vDash_{c} \langle x, N \rangle : \Sigma\, x{:}A\,.\,\underline{C}(x)}}{\Gamma \vDash_{c} M \;\texttt{to}\; x{:}A \;\texttt{in}\; \langle x, N \rangle : \Sigma\, x{:}A\,.\,\underline{C}(x)}$$

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

$$\frac{\Gamma \vDash_{v} \varepsilon_1 : \mathsf{Effect} \quad \Gamma \vdash A \quad \Gamma \vDash_{v} \varepsilon_2 : A \to \mathsf{Effect}}{\Gamma \vdash T\,\varepsilon_1\,A\,\varepsilon_2}$$

  in terms of standard parameterised effect-typing as

$$T\,\varepsilon_1\,A\,\varepsilon_2 \stackrel{\text{def}}{=} U_{\varepsilon_1}(\Sigma\, x{:}A\,.\,F_{\varepsilon_2\,x}\,1)$$

  and thus naturally accommodate examples like

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

Fibred adjunction models

(categorical semantics of eMLTT)

Fibred adjunction models – value part

Given by a split closed comprehension category p, as in

$$\begin{tikzcd} \mathcal{V} \arrow[d, bend right=60, "p"'] \arrow[d, phantom, shift right=4, "\dashv"] \arrow[d, phantom, shift left=4, "\dashv"] \arrow[d, bend left=60, "\{-\}"] \\ \mathcal{B} \arrow[u, "1"'] \end{tikzcd}$$

allowing us to define a partial interpretation fun. $[\![-]\!]$, that maps:

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

Fibred adjunction models – value part

Given by a split closed comprehension category p, as in

$$
\begin{tikzcd}
\mathcal{V} \arrow[d, bend right=60, "p"'] \arrow[d, bend right=30, phantom, "\dashv"] \arrow[d, bend left=60, "\{-\}"] \arrow[d, bend left=30, phantom, "\dashv"] \\
\mathcal{B} \arrow[u, "1"']
\end{tikzcd}
$$

such that

- p has split fibred strong colimits of shape $\mathbf{0}$ and $\mathbf{2}$ [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

$$
\begin{tikzcd}
\mathcal{V} \arrow[rr, bend left, "F"] \arrow[rr, phantom, "\perp"] \arrow[d, bend right=60, "p"'] \arrow[d, bend left=60, "\{-\}"] & & \mathcal{C} \arrow[ll, bend left, "U"] \arrow[dll, bend left, "q"] \\
\mathcal{B} \arrow[u, "1"] & &
\end{tikzcd}
$$

we extend the partial interpretation fun. $[\![-]\!]$ so that it maps:

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

$$[\![\Gamma; z : \underline{C}; K]\!] : [\![\Gamma; \underline{C}]\!] \longrightarrow \underline{D} \text{ in } \mathcal{C}_{[\![\Gamma]\!]}$$

Fibred adjunction models – effects part

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

$$\begin{tikzcd}
\mathcal{V} \arrow[rr, bend left, "F"] \arrow[rr, phantom, "\perp"] \arrow[d, bend right=50, "p"'] \arrow[d, bend left=50, "\{-\}"] & & \mathcal{C} \arrow[ll, bend left, "U"] \arrow[dll, bend left, "q"] \\
\mathcal{B} \arrow[u, "1"] & &
\end{tikzcd}$$

such that

- q has split dependent p-products (comp. Π-type; r. adj. to wk.)
- q has split dependent p-coproducts (comp. Σ-type; l. adj. to wk.)

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

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

Fibred adjunction models – correctness

Theorem (Soundness):

- If $\Gamma \vdash \underline{C}$, then $[\![\Gamma; \underline{C}]\!] \in \mathcal{C}_{[\![\Gamma]\!]}$
- If $\Gamma \vDash_{\mathsf{c}} M : \underline{C}$, then $[\![\Gamma; M]\!] : 1_{[\![\Gamma]\!]} \longrightarrow U([\![\Gamma; \underline{C}]\!])$
- If $\Gamma \,|\, z : \underline{C} \vDash_{\mathsf{h}} K : \underline{D}$, then $[\![\Gamma; z : \underline{C}; K]\!] : [\![\Gamma; \underline{C}]\!] \longrightarrow [\![\Gamma; \underline{D}]\!]$
- If $\Gamma \vdash \underline{C} = \underline{D}$, then $[\![\Gamma; \underline{C}]\!] = [\![\Gamma; \underline{D}]\!] \in \mathcal{C}_{[\![\Gamma]\!]}$
- ...

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

$$
\begin{tikzcd}[column sep=huge, row sep=huge]
\mathcal{V} \arrow[rr, bend left=15, "F"] \arrow[rr, phantom, "\perp"] \arrow[d, bend right=40, "p"'] \arrow[d, bend left=40, "\{-\}"] & & \mathcal{C} \arrow[ll, bend left=15, "U"] \arrow[dll, bend left=20, "q"] \\
\mathcal{B} \arrow[u, "1"'] \arrow[u, phantom, shift right=3ex, "\dashv"] \arrow[u, phantom, shift left=3ex, "\dashv"] & &
\end{tikzcd}
$$

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 $F \dashv U : \mathcal{C} \longrightarrow \mathcal{V}$ ($\mathcal{V}$ a CCC, ...),

  we can lift it to simple fibrations $\widehat{F} \dashv \widehat{U} : \mathsf{s}(\mathcal{V}, \mathcal{C}) \longrightarrow \mathsf{s}(\mathcal{V})$

  where

$$\mathsf{s}_{\mathcal{V},\mathcal{C}} : \mathsf{s}(\mathcal{V}, \mathcal{C}) \longrightarrow \mathcal{V}$$

  is defined as

$$\mathsf{s}_{\mathcal{V},\mathcal{C}}\Big(X \in \mathcal{V}\,,\, \underline{C} \in \mathcal{C}\Big) \overset{\mathrm{def}}{=} X$$

$$\mathsf{s}_{\mathcal{V},\mathcal{C}}\Big(f : X \longrightarrow Y\,;\, h : X \otimes \underline{C} \longrightarrow \underline{D}\Big) \overset{\mathrm{def}}{=} f \qquad : \mathsf{s}_{\mathcal{V},\mathcal{C}}(X, \underline{C}) \longrightarrow \mathsf{s}_{\mathcal{V},\mathcal{C}}(Y, \underline{D})$$

- 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 $F \dashv U : \mathcal{C} \longrightarrow \mathcal{V}$ ($\mathcal{V}$ a CCC, . . .),

  we can lift it to simple fibrations $\widehat{F} \dashv \widehat{U} : \mathsf{s}(\mathcal{V}, \mathcal{C}) \longrightarrow \mathsf{s}(\mathcal{V})$

  where

$$\mathsf{s}_{\mathcal{V},\mathcal{C}} : \mathsf{s}(\mathcal{V}, \mathcal{C}) \longrightarrow \mathcal{V}$$

  is defined as

$$\mathsf{s}_{\mathcal{V},\mathcal{C}}\Big(X \in \mathcal{V}\,,\,\underline{C} \in \mathcal{C}\Big) \stackrel{\text{def}}{=} X$$

$$\mathsf{s}_{\mathcal{V},\mathcal{C}}\Big(f : X \longrightarrow Y\,,\,h : X \otimes \underline{C} \longrightarrow \underline{D}\Big) \stackrel{\text{def}}{=} f \qquad : \mathsf{s}_{\mathcal{V},\mathcal{C}}(X, \underline{C}) \longrightarrow \mathsf{s}_{\mathcal{V},\mathcal{C}}(Y, \underline{D})$$

- 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 $F_{\mathcal{D}} \dashv U_{\mathcal{D}} : \mathcal{D} \longrightarrow \mathsf{Set}$,

  we can lift it to $\widehat{F_{\mathcal{D}}} \dashv \widehat{U_{\mathcal{D}}} : \mathsf{Fam}(\mathcal{D}) \longrightarrow \mathsf{Fam}(\mathsf{Set})$

  between
  $$\mathsf{fam}_{\mathsf{Set}} : \mathsf{Fam}(\mathsf{Set}) \longrightarrow \mathsf{Set}$$
  $$\mathsf{fam}_{\mathcal{D}} : \mathsf{Fam}(\mathcal{D}) \longrightarrow \mathsf{Set}$$

- resulting in
  - $(\llbracket\Gamma\rrbracket, \llbracket A\rrbracket) \in \mathsf{Fam}(\mathsf{Set})$ (where $\llbracket A\rrbracket \in \llbracket\Gamma\rrbracket \longrightarrow \mathsf{Set}$)
  - $(\llbracket\Gamma\rrbracket, \llbracket \underline{C}\rrbracket) \in \mathsf{Fam}(\mathcal{D})$ (where $\llbracket \underline{C}\rrbracket \in \llbracket\Gamma\rrbracket \longrightarrow \mathcal{D}$)

- examples
  - $F^{\mathbf{T}} \dashv U^{\mathbf{T}} : \mathsf{Set}^{\mathbf{T}} \longrightarrow \mathsf{Set}$
  - $(-) \times S \ \dashv \ (-)^S : \mathsf{Set} \longrightarrow \mathsf{Set}$
  - $R^{(-)} \ \dashv \ R^{(-)} : \mathsf{Set}^{op} \longrightarrow \mathsf{Set}$

Examples of fibred adjunction models

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

- given the EM-adjunction $F^{\mathbf{T}} \dashv U^{\mathbf{T}} : \mathrm{CPO}^{\mathbf{T}} \longrightarrow \mathrm{CPO}$,

  we can lift it to $\widehat{F_{\mathcal{D}}} \dashv \widehat{U_{\mathcal{D}}} : \mathrm{CFam}(\mathrm{CPO}^{\mathbf{T}}) \longrightarrow \mathrm{CFam}(\mathrm{CPO})$

  between

$$\mathrm{cfam}_{\mathrm{CPO}} : \mathrm{CFam}(\mathrm{CPO}) \longrightarrow \mathrm{CPO}$$

$$\mathrm{cfam}_{\mathrm{CPO}^{\mathbf{T}}} : \mathrm{CFam}(\mathrm{CPO}^{\mathbf{T}}) \longrightarrow \mathrm{CPO}$$

- resulting in
  - $(\llbracket\Gamma\rrbracket, \llbracket A\rrbracket) \in \mathrm{CFam}(\mathrm{CPO})$ (where $\llbracket A\rrbracket \in \llbracket\Gamma\rrbracket \longrightarrow \mathrm{CPO}^{EP}$)
  - $(\llbracket\Gamma\rrbracket, \llbracket \underline{C}\rrbracket) \in \mathrm{CFam}(\mathrm{CPO}^{\mathbf{T}})$ (where $\llbracket \underline{C}\rrbracket \in \llbracket\Gamma\rrbracket \longrightarrow (\mathrm{CPO}^{\mathbf{T}})^{EP}$)

- **Note:** $\mathrm{cod}_{\mathrm{CPO}}$ is not suitable because CPO is not an LCCC.

Examples of fibred adjunction models

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

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

$$\begin{tikzcd} \mathcal{V} \arrow[rr, "T"] \arrow[dr, "p"'] & & \mathcal{V} \arrow[dl, "p"] \\ & \mathcal{B} & \end{tikzcd}$$

and $\quad p(\eta_A) = \mathrm{id}_{p(A)} \qquad p(\mu_A) = \mathrm{id}_{p(A)}$

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

$$\begin{tikzcd} \mathcal{V} \arrow[rr, bend left, "F^{\mathbf{T}}"] \arrow[rr, phantom, "\perp"] \arrow[dr, "p"'] & & \mathcal{V}^{\mathbf{T}} \arrow[ll, bend left, "U^{\mathbf{T}}"] \arrow[dl, "p^{\mathbf{T}}"] \\ & \mathcal{B} & \end{tikzcd}$$

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

- 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 Π-types, then $p^{\mathbf{T}}$ also supports Π-types

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

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

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

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

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

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

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

Examples of fibred adjunction models

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

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

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

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

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

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

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

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

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

Examples of fibred adjunction models

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

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

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

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

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

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

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

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

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

Algebraic effects

(operations and equations)

Algebraic effects – ops. and eqs.

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

- signatures of dependently typed operation symbols

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

- equipped with equations on derivable effect terms

In eMLTT:

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

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

$$\frac{\Gamma \vdash_{v} V : I \quad \Gamma, x : O[V/x_i] \vdash_{c} M : \underline{C} \quad \Gamma \mid z : \underline{C} \vdash_{h} K : \underline{D}}{\Gamma \vdash_{c} K[\mathrm{op}^{\underline{C}}_{V}(x.M)/z] = \mathrm{op}^{\underline{D}}_{V}(x.K[M/z]) : \underline{D}} \ (\mathrm{op} : (x_i : I) \rightharpoonup O)$$

Sound semantics: Based on families fibrations and Law. theories

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

Algebraic effects – ops. and eqs.

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

- signatures of dependently typed operation symbols

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

- equipped with equations on derivable effect terms

In eMLTT:

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

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

$$\frac{\Gamma \vdash_{\mathsf{v}} V : I \quad \Gamma, x : O[V/x_i] \vdash_{\mathsf{c}} M : \underline{C} \quad \Gamma \,|\, z : \underline{C} \vdash_{\mathsf{h}} K : \underline{D}}{\Gamma \vdash_{\mathsf{c}} K[\mathsf{op}^{\underline{C}}_{V}(x.M)/z] = \mathsf{op}^{\underline{D}}_{V}(x.K[M/z]) : \underline{D}} \ (\mathsf{op} : (x_i : I) \rightharpoonup O)$$

Sound semantics: Based on families fibrations and Law. theories

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

Algebraic effects – ops. and eqs.

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

- signatures of dependently typed operation symbols

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

- equipped with equations on derivable effect terms

In eMLTT:

$$M \;::=\; \ldots \;\mid\; \mathsf{op}^{\underline{C}}_{V}(x.M)$$

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

$$\frac{\Gamma \vDash_{\mathsf{v}} V : I \quad \Gamma, x : O[V/x_i] \vDash_{\mathsf{c}} M : \underline{C} \quad \Gamma \,|\, z : \underline{C} \vDash_{\mathsf{h}} K : \underline{D}}{\Gamma \vDash_{\mathsf{c}} K[\mathsf{op}^{\underline{C}}_{V}(x.M)/z] = \mathsf{op}^{\underline{D}}_{V}\big(x.K[M/z]\big) : \underline{D}} \; (\mathsf{op} : (x_i : I) \rightharpoonup O)$$

Sound semantics: Based on families fibrations and Law. theories

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

Algebraic effects – ops. and eqs.

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

- signatures of dependently typed operation symbols

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

- equipped with equations on derivable effect terms

In eMLTT:

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

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

$$\frac{\Gamma \vDash_{\mathsf{v}} V : I \quad \Gamma, x : O[V/x_i] \vDash_{\mathsf{c}} M : \underline{C} \quad \Gamma \,|\, z : \underline{C} \vDash_{\mathsf{h}} K : \underline{D}}{\Gamma \vDash_{\mathsf{c}} K[\mathsf{op}^{\underline{C}}_{V}(x.M)/z] = \mathsf{op}^{\underline{D}}_{V}(x.K[M/z]) : \underline{D}} \ (\mathsf{op} : (x_i : I) \rightharpoonup O)$$

Sound semantics: Based on families fibrations and Law. theories

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

Algebraic effects – examples

Example 1 (interactive IO):

- $\mathsf{read} : 1 \rightharpoonup \mathsf{Chr}$ $\qquad (\mathsf{Chr} \overset{\text{def}}{=} 1 + \ldots + 1)$

  $\mathsf{write} : \mathsf{Chr} \rightharpoonup 1$

- no equations

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

- $\diamond \vdash \mathsf{Loc}$

  $\ell : \mathsf{Loc} \vdash \mathsf{Val}$

  $\diamond \vdash_{\mathsf{v}} \mathsf{isDec}_{\mathsf{Loc}} : \Pi\, \ell : \mathsf{Loc} \,.\, \Pi\, \ell' : \mathsf{Loc} \,.\, (\ell =_{\mathsf{Loc}} \ell') + (\ell =_{\mathsf{Loc}} \ell' \to 0)$

- $\mathsf{get} : (\ell : \mathsf{Loc}) \rightharpoonup \mathsf{Val}$

  $\mathsf{put} : (\Sigma\, \ell : \mathsf{Loc} \,.\, \mathsf{Val}) \rightharpoonup 1$

- five equations (two of them branching on $\mathsf{isDec}_{\mathsf{Loc}}$)

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 : $1 \rightharpoonup \mathsf{Chr}$ $\qquad$ $(\mathsf{Chr} \stackrel{\text{def}}{=} 1 + \ldots + 1)$

  write : $\mathsf{Chr} \rightharpoonup 1$

- no equations

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

- $\diamond \vdash \mathsf{Loc}$

  $\ell : \mathsf{Loc} \vdash \mathsf{Val}$

  $\diamond \vDash_{\mathsf{v}} \mathsf{isDec}_{\mathsf{Loc}} : \Pi\, \ell : \mathsf{Loc}\, .\, \Pi\, \ell' : \mathsf{Loc}\, .\, (\ell =_{\mathsf{Loc}} \ell') + (\ell =_{\mathsf{Loc}} \ell' \to 0)$

- get : $(\ell : \mathsf{Loc}) \rightharpoonup \mathsf{Val}$

  put : $(\Sigma\, \ell : \mathsf{Loc}\, .\mathsf{Val}) \rightharpoonup 1$

- five equations (two of them branching on $\mathsf{isDec}_{\mathsf{Loc}}$)

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

Algebraic effects – examples

Example 1 (interactive IO):

- $\mathsf{read} : 1 \rightharpoonup \mathsf{Chr}$ $\qquad (\mathsf{Chr} \stackrel{\text{def}}{=} 1 + \ldots + 1)$

  $\mathsf{write} : \mathsf{Chr} \rightharpoonup 1$

- no equations

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

- $\diamond \vdash \mathsf{Loc}$

  $\ell : \mathsf{Loc} \vdash \mathsf{Val}$

  $\diamond \vDash_{\mathsf{v}} \mathsf{isDec}_{\mathsf{Loc}} : \Pi\, \ell : \mathsf{Loc} \,.\, \Pi\, \ell' : \mathsf{Loc} \,.\, (\ell =_{\mathsf{Loc}} \ell') + (\ell =_{\mathsf{Loc}} \ell' \to 0)$

- $\mathsf{get} : (\ell : \mathsf{Loc}) \rightharpoonup \mathsf{Val}$

  $\mathsf{put} : (\Sigma\, \ell : \mathsf{Loc} \,.\, \mathsf{Val}) \rightharpoonup 1$

- five equations (two of them branching on $\mathsf{isDec}_{\mathsf{Loc}}$)

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

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:

$$\Gamma \vDash_{c} M \ \mathtt{handled\ with}\ \{\mathtt{op}_{x_v}(x_k) \mapsto N_{\mathsf{op}}\}_{\mathsf{op} \in \mathcal{T}_{\mathsf{eff}}}\ \mathtt{to}\ y\!:\!A\ \mathtt{in}_{\underline{C}}\ N_{\mathsf{ret}} : \underline{C}$$

satisfying

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

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

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:

$$\Gamma \models_{\mathsf{c}} M \;\texttt{handled with}\; \{\mathtt{op}_{x_v}(x_k) \mapsto N_{\mathsf{op}}\}_{\mathsf{op} \in \mathcal{T}_{\mathsf{eff}}} \;\texttt{to}\; y\!:\!A \;\texttt{in}_{\underline{C}}\; N_{\mathsf{ret}} : \underline{C}$$

satisfying

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

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

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:

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

satisfying

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

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

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:

$$\Gamma \vDash_{c} M \,\texttt{handled with}\, \{\mathtt{op}_{x_v}(x_k) \mapsto N_{\mathtt{op}}\}_{\mathtt{op} \in \mathcal{T}_{\mathrm{eff}}} \,\texttt{to}\, y\!:\!A \,\texttt{in}_{\underline{C}}\, N_{\mathrm{ret}} : \underline{C}$$

satisfying

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

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

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 reasoning

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

$$M \;\texttt{handled with}\; \{\mathtt{op}_{x_v}(x_k) \mapsto V_{\mathtt{op}}\}_{\mathtt{op} \in \mathcal{T}_{\mathsf{eff}}} \;\texttt{to}\; y : A \;\texttt{in}_{\underline{B}}\; V_{\mathsf{ret}}$$

we can define natural predicates (essentially, dependent types)

$$\Gamma \vDash_{\mathsf{v}} P : UFA \to \mathcal{U}$$

by

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

Note 1: $P\,(\texttt{thunk}\,M)$ 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, $+$, Σ, and Π
- a type-based treatment of handlers $\quad \underline{C} \;::=\; \ldots \mid \langle A ; \overrightarrow{V_{\mathsf{op}}} ; \overrightarrow{W_{\mathsf{eq}}} \rangle$
- function extensionality $\qquad$ (actually, it's a bit more extensional)

Handlers of alg. effects – for reasoning

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

$$M \;\texttt{handled with}\; \{\mathtt{op}_{x_v}(x_k) \mapsto V_{\mathtt{op}}\}_{\mathtt{op} \in \mathcal{T}_{\mathrm{eff}}} \;\texttt{to}\; y\!:\!A \;\texttt{in}_{B}\; V_{\mathrm{ret}}$$

we can define natural predicates (essentially, dependent types)

$$\Gamma \vDash_{v} P : UFA \to \mathcal{U}$$

by

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

Note 1: $P\,(\texttt{thunk}\,M)$ 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, $+$, Σ, and Π
- a type-based treatment of handlers $\quad \underline{C} \;::=\; \ldots \mid \langle A; \overrightarrow{V_{\mathrm{op}}}; \overrightarrow{W_{\mathrm{eq}}} \rangle$
- function extensionality $\qquad$ (actually, it's a bit more extensional)

Handlers of alg. effects – for reasoning

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

$$M \texttt{ handled with } \{\mathtt{op}_{x_v}(x_k) \mapsto V_{\mathrm{op}}\}_{\mathrm{op}\in\mathcal{T}_{\mathrm{eff}}} \texttt{ to } y\!:\!A \texttt{ in}_{B}\ V_{\mathrm{ret}}$$

we can define natural predicates (essentially, dependent types)

$$\Gamma \models_{\mathsf{v}} P : UFA \to \mathcal{U}$$

by

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

Note 1: $P\,(\mathtt{thunk}\,M)$ 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, $+$, Σ, and Π
- a type-based treatment of handlers $\quad \underline{C} ::= \ldots \mid \langle A; \overrightarrow{V_{\mathrm{op}}}; \overrightarrow{W_{\mathrm{eq}}} \rangle$
- function extensionality $\qquad$ (actually, it's a bit more extensional)

Handlers of alg. effects – for reasoning

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

$$M \;\texttt{handled with}\; \{\mathsf{op}_{x_v}(x_k) \mapsto V_{\mathsf{op}}\}_{\mathsf{op} \in \mathcal{T}_{\mathsf{eff}}} \;\texttt{to}\; y\!:\!A \;\texttt{in}_{B}\; V_{\mathsf{ret}}$$

we can define natural predicates (essentially, dependent types)

$$\Gamma \vDash_{v} P : UFA \to \mathcal{U}$$

by

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

Note 1: $P\,(\texttt{thunk}\, M)$ 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, $+$, Σ, and Π
- a type-based treatment of handlers $\quad \underline{C} \;::=\; \ldots \mid \langle A; \overrightarrow{V_{\mathsf{op}}}; \overrightarrow{W_{\mathsf{eq}}} \rangle$
- function extensionality (actually, it's a bit more extensional)

Handlers of alg. effects – for reasoning

Example 1 (Evaluation Logic style modalities):

- Given a predicate $P : A \to \mathcal{U}$ on return values,

  we define a predicate $\Diamond P : UFA \to \mathcal{U}$ on IO-computations as

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

  using the handler given by

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

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

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

$$\Diamond P\, (\mathtt{thunk}\, (\mathtt{read}(x \,.\, \mathtt{write}_{e'}(\mathtt{return}\, V)))) = \widehat{\Sigma}\, x \,{:}\, \mathsf{El}(\widehat{\mathsf{Chr}}) \,.\, P\ V$$

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

Handlers of alg. effects – for reasoning

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

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

  using the handler given by

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

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

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

$$\Diamond P\,(\texttt{thunk}\,(\texttt{read}(x\,.\,\texttt{write}_{e'}(\texttt{return}\, V)))) = \widehat{\Sigma}\, x \,{:}\, \mathsf{El}(\widehat{\mathsf{Chr}})\,.\,P\ V$$

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

Handlers of alg. effects – for reasoning

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

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

  using the handler given by

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

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

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

$$\Diamond P\,\big(\texttt{thunk}\,(\texttt{read}(x\,.\,\texttt{write}_{e'}(\texttt{return}\, V)))\big) = \widehat{\Sigma}\, x : \mathsf{El}(\widehat{\mathsf{Chr}})\,.\,P\ V$$

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

Handlers of alg. effects – for reasoning

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

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

  using the handler given by

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

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

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

$$\Diamond P\,\big(\texttt{thunk}\,(\texttt{read}(x\,.\,\texttt{write}_{e'}(\texttt{return}\, V)))\big) = \widehat{\Sigma}\, x\!:\!\mathsf{El}(\widehat{\mathsf{Chr}})\,.\,P\ V$$

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

  $$Q : A \to S \to \mathcal{U} \qquad\qquad (S \stackrel{\text{def}}{=} \Pi\, \ell : \mathsf{Loc}\,.\,\mathsf{Val})$$

  we define a precondition for stateful comps. on initial states

  $$\mathsf{wp}_Q : UFA \to S \to \mathcal{U}$$

  by

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

  $$V_{\mathsf{get}}, V_{\mathsf{put}} \quad \text{on} \quad S \to (\mathcal{U} \times S) \qquad \text{and} \qquad V_{\mathsf{ret}} \text{ ``$=$'' } Q$$

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

- **Theorem:** wp_Q satisfies expected properties of WPs, e.g.,

  $$\mathsf{wp}_Q\,(\mathtt{thunk}\,(\mathtt{return}\,V)) \quad = \quad \lambda\, x_S : S\,.\,Q\ V\ x_S$$

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

Handlers of alg. effects – for reasoning

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

- Given a postcondition on return values and final states

$$Q : A \to S \to \mathcal{U} \qquad\qquad (S \stackrel{\text{def}}{=} \Pi\, \ell : \mathsf{Loc} \,.\, \mathsf{Val})$$

  we define a precondition for stateful comps. on initial states

$$\mathsf{wp}_Q : UFA \to S \to \mathcal{U}$$

  by

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

$$V_{\mathsf{get}}, V_{\mathsf{put}} \quad \text{on} \quad S \to (\mathcal{U} \times S) \qquad \text{and} \qquad V_{\mathsf{ret}} \text{ ``$=$'' } Q$$

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

- **Theorem:** wp_Q satisfies expected properties of WPs, e.g.,

$$\mathsf{wp}_Q\,(\mathtt{thunk}\,(\mathtt{return}\,V)) \;=\; \lambda\, x_S : S \,.\, Q\; V\; x_S$$

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

Handlers of alg. effects – for reasoning

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

- Given a postcondition on return values and final states

  $$Q : A \to S \to \mathcal{U} \qquad\qquad (S \stackrel{\text{def}}{=} \Pi\, \ell : \mathsf{Loc} \,.\, \mathsf{Val})$$

  we define a precondition for stateful comps. on initial states

  $$\mathsf{wp}_Q : UFA \to S \to \mathcal{U}$$

  by

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

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

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

- **Theorem:** wp_Q satisfies expected properties of WPs, e.g.,

  $$\mathsf{wp}_Q\, (\mathtt{thunk}\, (\mathtt{return}\, V)) \quad = \quad \lambda\, x_S : S \,.\, Q\ V\ x_S$$

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

Handlers of alg. effects – for reasoning

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

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

$$\mathtt{e} : \mathsf{Protocol} \qquad \mathtt{r} : (\mathsf{Chr} \to \mathsf{Protocol}) \to \mathsf{Protocol}$$

$$\mathtt{w} : (\mathsf{Chr} \to \mathcal{U}) \to \mathsf{Protocol} \to \mathsf{Protocol}$$

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

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

$$\mathsf{Allowed} : UFA \to \mathsf{Protocol} \to \mathcal{U}$$

by handling the given computation using

$$V_{\mathsf{read}}\,,\, V_{\mathsf{write}} \qquad \text{on} \qquad \mathsf{Protocol} \to \mathcal{U}$$

where

$$\begin{array}{lcl}
V_{\mathsf{read}}\ \langle _\,,\, V_{\mathsf{rk}} \rangle\ (\mathtt{r}\ \mathsf{Pr}') & \stackrel{\mathsf{def}}{=} & \widehat{\Pi}\, x \mathbin{:} \mathsf{El}(\widehat{\mathsf{Chr}})\,.\,(V_{\mathsf{rk}}\ x)\ (\mathsf{Pr}'\ x) \\
V_{\mathsf{write}}\ \langle V\,,\, V_{\mathsf{wk}} \rangle\ (\mathtt{w}\ P\ \mathsf{Pr}') & \stackrel{\mathsf{def}}{=} & \widehat{\Sigma}\, x \mathbin{:} \mathsf{El}(P\ V)\,.\,V_{\mathsf{wk}}\ \star\ \mathsf{Pr}' \\
_ & \stackrel{\mathsf{def}}{=} & \widehat{0}
\end{array}$$

Handlers of alg. effects – for reasoning

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

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

$$\mathtt{e} : \mathsf{Protocol} \qquad\qquad \mathtt{r} : (\mathsf{Chr} \to \mathsf{Protocol}) \to \mathsf{Protocol}$$

$$\mathtt{w} : (\mathsf{Chr} \to \mathcal{U}) \to \mathsf{Protocol} \to \mathsf{Protocol}$$

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

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

$$\mathsf{Allowed} : UFA \to \mathsf{Protocol} \to \mathcal{U}$$

  by handling the given computation using

$$V_{\mathsf{read}}, V_{\mathsf{write}} \qquad \text{on} \qquad \mathsf{Protocol} \to \mathcal{U}$$

  where

$$\begin{array}{lcl} V_{\mathsf{read}}\ \langle \_ , V_{\mathsf{rk}} \rangle\ (\mathtt{r}\ \mathsf{Pr}') & \stackrel{\mathsf{def}}{=} & \widehat{\Pi}\, x : \mathsf{El}(\widehat{\mathsf{Chr}})\,.\,(V_{\mathsf{rk}}\ x)\ (\mathsf{Pr}'\ x) \\ V_{\mathsf{write}}\ \langle V , V_{\mathsf{wk}} \rangle\ (\mathtt{w}\ P\ \mathsf{Pr}') & \stackrel{\mathsf{def}}{=} & \widehat{\Sigma}\, x : \mathsf{El}(P\ V)\,.\,V_{\mathsf{wk}}\ \star\ \mathsf{Pr}' \\ \_ & \stackrel{\mathsf{def}}{=} & \widehat{0} \end{array}$$

Handlers of alg. effects – for reasoning

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

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

$$\mathtt{e} : \mathsf{Protocol} \qquad \mathtt{r} : (\mathsf{Chr} \to \mathsf{Protocol}) \to \mathsf{Protocol}$$

$$\mathtt{w} : (\mathsf{Chr} \to \mathcal{U}) \to \mathsf{Protocol} \to \mathsf{Protocol}$$

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

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

$$\mathsf{Allowed} : UFA \to \mathsf{Protocol} \to \mathcal{U}$$

  by handling the given computation using

$$V_{\mathsf{read}}, V_{\mathsf{write}} \qquad \text{on} \qquad \mathsf{Protocol} \to \mathcal{U}$$

  where

$$\begin{array}{lll} V_{\mathsf{read}} \ \langle \_ \,, V_{\mathsf{rk}} \rangle \ (\mathtt{r}\ \mathsf{Pr}') & \stackrel{\mathsf{def}}{=} & \widehat{\Pi}\, x\!:\!\mathsf{El}(\widehat{\mathsf{Chr}})\,.\,(V_{\mathsf{rk}}\ x)\ (\mathsf{Pr}'\ x) \\ V_{\mathsf{write}} \ \langle V \,, V_{\mathsf{wk}} \rangle \ (\mathtt{w}\ P\ \mathsf{Pr}') & \stackrel{\mathsf{def}}{=} & \widehat{\Sigma}\, x\!:\!\mathsf{El}(P\ V)\,.\,V_{\mathsf{wk}} \ \star \ \mathsf{Pr}' \\ \_ & \stackrel{\mathsf{def}}{=} & \widehat{0} \end{array}$$

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.
Fibred Computational Effects. (PhD Thesis, 2017)

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

D. Ahman.
Handling Fibred Computational Effects. (POPL'18)