Dependent Types and Fibred Computational Effects

Danel Ahman[1]

(joint work with Neil Ghani[2] and Gordon Plotkin[1])

[1]LFCS, University of Edinburgh

[2]MSP Group, University of Strathclyde

April 4, 2016

Outline

Language design principles for combining

- dependent types $(\Pi, \Sigma, \mathsf{Id}_A(V, W), \ldots)$
- computational effects (state, I/O, probability, recursion, ...)

Our goal

- have a mathematically natural story
- use established math. techniques
- cover a wide range of computational effects

This work was guided by two problems

- effectful programs in types
- assigning types to effectful programs

Effectful programs in types

(type-dependency in the presence of effects)

Effectful programs in types

Let's assume that we have a dependent type $A(x)$, e.g.:

$$x : \mathsf{Nat} \vdash A(x) \quad \stackrel{\text{def}}{=} \quad \texttt{if}\ (x\ \texttt{mod}\ 2 == 0)\ \texttt{then}\ \mathsf{String}\ \texttt{else}\ \mathsf{Char}$$

Q: Should we allow $A[M/x]$ if M is an effectful program?

- e.g., if M is $\mathsf{receive}(y.N)$

A1: In this work we say no

- types should only depend on static information
- e.g., how would one compute $A[\mathsf{receive}(y.M)/x]$ statically?
- we recover dependency on effectful computations via thunks

A2: In a separate line of work, we are also looking at yes

- type-dependency ($z : \underline{C} \vdash A(z)$) becomes "homomorphic"
- lifting effect operations from terms to types, e.g., $\langle \mathsf{receive} \rangle (y.\, A)$
- similarities with refinement types and op. modalities [A.,P.'15]

Effectful programs in types

Let's assume that we have a dependent type $A(x)$, e.g.:

$$x\!:\!\mathsf{Nat} \vdash A(x) \quad \overset{\text{def}}{=} \quad \texttt{if}\ (x\ \texttt{mod}\ 2 == 0)\ \texttt{then}\ \mathsf{String}\ \texttt{else}\ \mathsf{Char}$$

Q: Should we allow $A[M/x]$ if M is an effectful program?

- e.g., if M is $\mathsf{receive}(y.N)$

A1: In this work we say no

- types should only depend on static information
- e.g., how would one compute $A[\mathsf{receive}(y.M)/x]$ statically?
- we recover dependency on effectful computations via thunks

A2: In a separate line of work, we are also looking at yes

- type-dependency ($z\!:\!\underline{C} \vdash A(z)$) becomes "homomorphic"
- lifting effect operations from terms to types, e.g., $\langle\mathsf{receive}\rangle(y.\,A)$
- similarities with refinement types and op. modalities [A.,P.'15]

Effectful programs in types

Let's assume that we have a dependent type $A(x)$, e.g.:

$$x\!:\!\mathsf{Nat} \vdash A(x) \quad \overset{\mathsf{def}}{=} \quad \texttt{if}\ (x\ \texttt{mod}\ 2 == 0)\ \texttt{then}\ \mathsf{String}\ \texttt{else}\ \mathsf{Char}$$

Q: Should we allow $A[M/x]$ if M is an effectful program?

- e.g., if M is $\mathsf{receive}(y.N)$

A1: In this work we say no

- types should only depend on static information
- e.g., how would one compute $A[\mathsf{receive}(y.M)/x]$ statically?
- we recover dependency on effectful computations via thunks

A2: In a separate line of work, we are also looking at yes

- type-dependency $(\, z\!:\!\underline{C} \vdash A(z)\,)$ becomes "homomorphic"
- lifting effect operations from terms to types, e.g., $\langle\mathsf{receive}\rangle(y.\,A)$
- similarities with refinement types and op. modalities [A.,P.'15]

Effectful programs in types ctd.

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. version of CBPV/EEC)
- where Γ contains **only** value variables $x_1 : A_1, \ldots, x_n : A_n$

Note: Some of the other options are λ_{ML} and FGCBV

- but basing the work on CBPV/EEC gives a more general story
- especially for treating of sequential composition
- also for systematically integrating dependent- and effect-typing (ongoing work)

Effectful programs in types ctd.

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. version of CBPV/EEC)
- where Γ contains **only** value variables $x_1 : A_1, \ldots, x_n : A_n$

Note: Some of the other options are λ_{ML} and FGCBV

- but basing the work on CBPV/EEC gives a more general story
- especially for treating of sequential composition
- also for systematically integrating dependent- and effect-typing (ongoing work)

Effectful programs in types ctd.

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. version of CBPV/EEC)
- where Γ contains **only** value variables $x_1 \,{:}\, A_1, \ldots, x_n \,{:}\, A_n$

Note: Some of the other options are λ_{ML} and FGCBV

- but basing the work on CBPV/EEC gives a more general story
- especially for treating of sequential composition
- also for systematically integrating dependent- and effect-typing (ongoing work)

Assigning types to effectful programs

(i.e., typing sequential composition)

Assigning types to effectful programs

The problem: The standard typing rule for seq. composition

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

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

$$\underline{C}$$

in the conclusion

Assigning types to effectful programs ctd.

Aim: To fix the typing rule of sequential composition

Option 1: We could restrict the free variables in $\underline{C}$, i.e.,

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

But sometimes it is necessary for $\underline{C}$ to depend on x!

- e.g., even to write effectful programs modularly
- take monadic parsing of well-typed syntax

  and consider writing a parser for function application
- it is natural to modularly decompose the code into

$$\cdot \vdash_{\mathsf{c}} \texttt{parseFun} : F\,(\Sigma y_1\!:\!\mathsf{LangType}.\Sigma y_2\!:\!\mathsf{LangType}.\mathsf{LangSyntax}(\texttt{fun}\; y_1\; y_2))$$

$$x\!:\!\Sigma y_1.\Sigma y_2.\mathsf{LangSyntax}(\texttt{fun}\; y_1\; y_2) \vdash_{\mathsf{c}} \texttt{parseFunArg} : F\,(\mathsf{LangSyntax}(\texttt{fst}\; x))$$

Assigning types to effectful programs ctd.

Aim: To fix the typing rule of sequential composition

Option 1: We could restrict the free variables in $\underline{C}$, i.e.,

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

But sometimes it is necessary for $\underline{C}$ to depend on x!

- e.g., even to write effectful programs modularly
- take monadic parsing of well-typed syntax

  and consider writing a parser for function application
- it is natural to modularly decompose the code into

$$\cdot \vDash_{\mathsf{c}} \texttt{parseFun} : F\,(\Sigma y_1\!:\!\mathsf{LangType}.\Sigma y_2\!:\!\mathsf{LangType}.\mathsf{LangSyntax}(\texttt{fun}\,y_1\,y_2))$$

$$x\!:\!\Sigma y_1.\Sigma y_2.\mathsf{LangSyntax}(\texttt{fun}\,y_1\,y_2) \vDash_{\mathsf{c}} \texttt{parseFunArg} : F\,(\mathsf{LangSyntax}(\texttt{fst}\,x))$$

Assigning types to effectful programs ctd.

Aim: To fix the typing rule of sequential composition

Option 1: We could restrict the free variables in $\underline{C}$, i.e.,

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

But sometimes it is necessary for $\underline{C}$ to depend on x!

- e.g., even to write effectful programs modularly
- take monadic parsing of well-typed syntax

  and consider writing a parser for function application
- it is natural to modularly decompose the code into

$$\cdot \vDash_{c} \texttt{parseFun} : F\,(\Sigma y_1\!:\!\mathsf{LangType}.\Sigma y_2\!:\!\mathsf{LangType}.\mathsf{LangSyntax}(\texttt{fun}\,y_1\,y_2))$$

$$x\!:\!\Sigma y_1.\Sigma y_2.\mathsf{LangSyntax}(\texttt{fun}\,y_1\,y_2) \vDash_{c} \texttt{parseFunArg} : F\,(\mathsf{LangSyntax}(\texttt{fst}\,x))$$

Assigning types to effectful programs ctd.

Aim: To fix the typing rule of sequential composition

Option 1: We could restrict the free variables in $\underline{C}$, i.e.,

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

But sometimes it is necessary for $\underline{C}$ to depend on x!

- e.g., even to write effectful programs modularly
- take monadic parsing of well-typed syntax

  and consider writing a parser for function application
- it is natural to modularly decompose the code into

$$\cdot \vDash_{c} \mathtt{parseFun} : F\,(\Sigma y_1\!:\!\mathrm{LangType}.\Sigma y_2\!:\!\mathrm{LangType}.\mathrm{LangSyntax}(\mathtt{fun}\,y_1\,y_2))$$

$$x\!:\!\Sigma y_1.\Sigma y_2.\mathrm{LangSyntax}(\mathtt{fun}\,y_1\,y_2) \vDash_{c} \mathtt{parseFunArg} : F\,(\mathrm{LangSyntax}(\mathtt{fst}\,x))$$

Assigning types to effectful programs ctd.

Aim: To fix the typing rule of sequential composition

Option 1: We could restrict the free variables in $\underline{C}$, i.e.,

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

But sometimes it is necessary for $\underline{C}$ to depend on x!

- e.g., even to write effectful programs modularly
- take monadic parsing of well-typed syntax

  and consider writing a parser for function application
- it is natural to modularly decompose the code into

$$\cdot \vDash_{\mathsf{c}} \mathtt{parseFun} : F\,(\Sigma y_1\!:\!\mathsf{LangType}.\Sigma y_2\!:\!\mathsf{LangType}.\mathsf{LangSyntax}(\mathtt{fun}\,y_1\,y_2))$$

$$x\!:\!\Sigma y_1.\Sigma y_2.\mathsf{LangSyntax}(\mathtt{fun}\,y_1\,y_2) \vDash_{\mathsf{c}} \mathtt{parseFunArg} : F\,(\mathsf{LangSyntax}(\mathtt{fst}\,x))$$

Assigning types to effectful programs ctd.

Aim: To fix the typing rule of sequential composition

Option 1: We could restrict the free variables in $\underline{C}$, i.e.,

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

But sometimes it is necessary for $\underline{C}$ to depend on x!

- e.g., even to write effectful programs modularly
- take monadic parsing of well-typed syntax

  and consider writing a parser for function application
- it is natural to modularly decompose the code into

$$\cdot \vdash_{\mathsf{c}} \texttt{parseFun} : F\,(\Sigma y_1\!:\!\mathsf{LangType}.\Sigma y_2\!:\!\mathsf{LangType}.\mathsf{LangSyntax}(\texttt{fun}\,y_1\,y_2))$$

$$x\!:\!\Sigma y_1.\Sigma y_2.\mathsf{LangSyntax}(\texttt{fun}\,y_1\,y_2) \vdash_{\mathsf{c}} \texttt{parseFunArg} : F\,(\mathsf{LangSyntax}(\texttt{fst}\,x))$$

Assigning types to effectful programs ctd.

Aim: To fix the typing rule of sequential composition

Option 1: We could restrict the free variables in $\underline{C}$, i.e.,

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

But sometimes it is necessary for $\underline{C}$ to depend on x!

- e.g., even to write effectful programs modularly
- take monadic parsing of well-typed syntax

  and consider writing a parser for function application
- it is natural to modularly decompose the code into

$$\cdot \vdash_c \texttt{parseFun} : F\,(\Sigma y_1\!:\!\mathsf{LangType}.\Sigma y_2\!:\!\mathsf{LangType}.\mathsf{LangSyntax}(\texttt{fun}\,y_1\,y_2))$$

$$x\!:\!\Sigma y_1.\Sigma y_2.\mathsf{LangSyntax}(\texttt{fun}\,y_1\,y_2) \vdash_c \texttt{parseFunArg} : F\,(\mathsf{LangSyntax}(\texttt{fst}\,x))$$

Assigning types to effectful programs ctd.

Aim: To fix the typing rule of sequential composition

Option 2: One could lift sequential composition to type level

$$\Gamma \vdash_{\mathsf{c}} M \;\texttt{to}\; x{:}A \;\texttt{in}\; N : M \;\texttt{to}\; x{:}A \;\texttt{in}\; \underline{C}$$

But then all comp. types would be singleton-like

- comp. types would contain exactly the terms we want to type!

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

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

But what makes this a principled solution? Why is it correct?

Assigning types to effectful programs ctd.

Aim: To fix the typing rule of sequential composition

Option 2: One could lift sequential composition to type level

$$\Gamma \vdash_{c} M \ \mathtt{to}\ x \,{:}\, A \ \mathtt{in}\ N : M \ \mathtt{to}\ x \,{:}\, A \ \mathtt{in}\ \underline{C}$$

But then all comp. types would be singleton-like

- comp. types would contain exactly the terms we want to type!

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

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

But what makes this a principled solution? Why is it correct?

Assigning types to effectful programs ctd.

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 all comp. types would be singleton-like

- comp. types would contain exactly the terms we want to type!

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

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

But what makes this a principled solution? Why is it correct?

Assigning types to effectful programs ctd.

Aim: To fix the typing rule of sequential composition

Option 4: We draw inspiration from algebraic effects

- and combine it with Option 1, i.e., restricting $\underline{C}$ in seq. comp.

E.g., consider the stateful program (for some $x : \mathsf{Nat} \vdash_{\mathsf{c}} N : \underline{C}$)

$$M \quad \stackrel{\text{def}}{=} \quad \texttt{lookup}\,(\texttt{return}\,2, \texttt{return}\,3)\ \texttt{to}\ x : \mathsf{Nat}\ \texttt{in}\ N$$

After looking up the bit, this program evaluates as either

$$N[2/x] \ \text{ at type } \ \underline{C}[2/x] \qquad \text{or} \qquad N[3/x] \ \text{ at type } \ \underline{C}[3/x]$$

Idea: M denotes an element of the coproduct of algebras

$$\underline{C}[2/x] + \underline{C}[3/x] \quad \stackrel{\text{def}}{=} \quad F\Big(U\left(\underline{C}[2/x]\right) + U\left(\underline{C}[3/x]\right)\Big)_{/\equiv}$$

- actually, we use a Nat-indexed coproduct (i.e., $\Sigma x : \mathsf{Nat}.\, \underline{C}$)

Assigning types to effectful programs ctd.

Aim: To fix the typing rule of sequential composition

Option 4: We draw inspiration from algebraic effects

- and combine it with Option 1, i.e., restricting $\underline{C}$ in seq. comp.

E.g., consider the stateful program (for some $x\!:\!\mathsf{Nat} \vdash_{\mathsf{c}} N : \underline{C}$)

$$M \overset{\text{def}}{=} \texttt{lookup}\,(\texttt{return}\,2, \texttt{return}\,3)\ \texttt{to}\ x\!:\!\mathsf{Nat}\ \texttt{in}\ N$$

After looking up the bit, this program evaluates as either

$$N[2/x] \text{ at type } \underline{C}[2/x] \qquad \text{or} \qquad N[3/x] \text{ at type } \underline{C}[3/x]$$

Idea: M denotes an element of the coproduct of algebras

$$\underline{C}[2/x] + \underline{C}[3/x] \overset{\text{def}}{=} F\Big(U\left(\underline{C}[2/x]\right) + U\left(\underline{C}[3/x]\right)\Big)_{/\equiv}$$

- actually, we use a Nat-indexed coproduct (i.e., $\Sigma x\!:\!\mathsf{Nat}.\,\underline{C}$)

Assigning types to effectful programs ctd.

Aim: To fix the typing rule of sequential composition

Option 4: We draw inspiration from algebraic effects

- and combine it with Option 1, i.e., restricting $\underline{C}$ in seq. comp.

E.g., consider the stateful program (for some $x\,{:}\,\mathsf{Nat} \vdash_{\mathsf{c}} N : \underline{C}$)

$$M \overset{\text{def}}{=} \mathtt{lookup}\,(\mathtt{return}\,2, \mathtt{return}\,3)\ \mathtt{to}\ x\,{:}\,\mathsf{Nat}\ \mathtt{in}\ N$$

After looking up the bit, this program evaluates as either

$$N[2/x] \ \text{ at type } \ \underline{C}[2/x] \qquad \text{or} \qquad N[3/x] \ \text{ at type } \ \underline{C}[3/x]$$

Idea: M denotes an element of the coproduct of algebras

$$\underline{C}[2/x] + \underline{C}[3/x] \overset{\text{def}}{=} F\Big(U\,(\underline{C}[2/x]) + U\,(\underline{C}[3/x])\Big)_{/\equiv}$$

- actually, we use a Nat-indexed coproduct (i.e., $\Sigma x\,{:}\,\mathsf{Nat}.\,\underline{C}$)

Assigning types to effectful programs ctd.

Aim: To fix the typing rule of sequential composition

Option 4: We draw inspiration from algebraic effects

- and combine it with Option 1, i.e., restricting $\underline{C}$ in seq. comp.

E.g., consider the stateful program (for some $x : \mathsf{Nat} \vdash_{\mathsf{c}} N : \underline{C}$)

$$M \quad \overset{\text{def}}{=} \quad \mathtt{lookup}\,(\mathtt{return}\,2, \mathtt{return}\,3)\ \mathtt{to}\ x : \mathsf{Nat}\ \mathtt{in}\ N$$

After looking up the bit, this program evaluates as either

$$N[2/x] \ \text{ at type } \ \underline{C}[2/x] \qquad \text{or} \qquad N[3/x] \ \text{ at type } \ \underline{C}[3/x]$$

Idea: M denotes an element of the coproduct of algebras

$$\underline{C}[2/x] + \underline{C}[3/x] \quad \overset{\text{def}}{=} \quad F\Big(U\,(\underline{C}[2/x]) + U\,(\underline{C}[3/x])\Big)_{/\equiv}$$

- actually, we use a Nat-indexed coproduct (i.e., $\Sigma x : \mathsf{Nat}.\, \underline{C}$)

Assigning types to effectful programs ctd.

Aim: To fix the typing rule of sequential composition

Option 4: We draw inspiration from algebraic effects

- and combine it with Option 1, i.e., restricting $\underline{C}$ in seq. comp.

E.g., consider the stateful program (for some $x : \mathsf{Nat} \vDash_{\mathsf{c}} N : \underline{C}$)

$$M \overset{\text{def}}{=} \mathtt{lookup}\,(\mathtt{return}\,2, \mathtt{return}\,3)\ \mathtt{to}\ x : \mathsf{Nat}\ \mathtt{in}\ N$$

After looking up the bit, this program evaluates as either

$$N[2/x] \ \text{ at type } \ \underline{C}[2/x] \qquad \text{or} \qquad N[3/x] \ \text{ at type } \ \underline{C}[3/x]$$

Idea: M denotes an element of the coproduct of algebras

$$\underline{C}[2/x] + \underline{C}[3/x] \overset{\text{def}}{=} F\left(U\left(\underline{C}[2/x]\right) + U\left(\underline{C}[3/x]\right)\right)_{/\equiv}$$

- actually, we use a Nat-indexed coproduct (i.e., $\Sigma x : \mathsf{Nat}.\, \underline{C}$)

Assigning types to effectful programs ctd.

Aim: To fix the typing rule of sequential composition

Option 4: We draw inspiration from algebraic effects

- and combine it with Option 1, i.e., restricting $\underline{C}$ in seq. comp.

E.g., consider the stateful program (for some $x : \mathsf{Nat} \vDash_{\mathsf{c}} N : \underline{C}$)

$$M \quad \overset{\text{def}}{=} \quad \mathtt{lookup}\,(\mathtt{return}\,2, \mathtt{return}\,3)\ \mathtt{to}\ x : \mathsf{Nat}\ \mathtt{in}\ N$$

After looking up the bit, this program evaluates as either

$$N[2/x] \ \text{ at type } \ \underline{C}[2/x] \qquad \text{or} \qquad N[3/x] \ \text{ at type } \ \underline{C}[3/x]$$

Idea: M denotes an element of the coproduct of algebras

$$\underline{C}[2/x] + \underline{C}[3/x] \quad \overset{\text{def}}{=} \quad F\left(U\left(\underline{C}[2/x]\right) + U\left(\underline{C}[3/x]\right)\right)_{/\equiv}$$

- actually, we use a Nat-indexed coproduct (i.e., $\Sigma x : \mathsf{Nat}.\, \underline{C}$)

Putting these ideas together

(a core dependently-typed calculus with comp. effects)

A computational dep.-typed language

Recall: We aim to define a dependently-typed language with

- general computational effects
- a clear distinction between
  - values
  - computations
- with a principled treatment of sequential composition
  - restricting free variables in seq. composition
  - based on coproducts of algebras
- with a natural denotational semantics, using standard techniques
  - dep. types — comprehension categories
  - comp. effects — adjunction models

A computational dep.-typed language

Value types: MLTT's types + thunks + ...

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

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

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

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

- $\Pi x : A.\underline{C}$ is the type of dependent effectful functions
  - it generalises CBPV's and EEC's computational function type $A \to \underline{C}$ and product type $\underline{C} \times \underline{D}$
- $\Sigma x : A.\underline{C}$ is the generalisation of coproducts of algebras
  - it generalises EEC's computational tensor type $A \otimes \underline{C}$ and sum type $\underline{C} + \underline{D}$

A computational dep.-typed language

Value types: MLTT's types + thunks + ...

$$A, B ::= \mathsf{Nat} \mid 1 \mid \Pi x\!:\!A.B \mid \Sigma x\!:\!A.B \mid Id_A(V, W) \mid U\,\underline{C} \mid \ldots$$

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

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

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

- $\Pi x\!:\!A.\underline{C}$ is the type of dependent effectful functions
    - it generalises CBPV's and EEC's computational function type $A \to \underline{C}$ and product type $\underline{C} \times \underline{D}$

- $\Sigma x\!:\!A.\underline{C}$ is the generalisation of coproducts of algebras
    - it generalises EEC's computational tensor type $A \otimes \underline{C}$ and sum type $\underline{C} + \underline{D}$

A computational dep.-typed language

Value terms: MLTT's terms + thunks + ...

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

- equational theory based on MLTT with intensional id.-types
- value terms are typed using a judgment $\Gamma \vdash_{\mathsf{v}} V : A$

Computation terms: dep.-typed version of CBPV/EEC c. terms

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

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

A computational dep.-typed language

Value terms: MLTT's terms + thunks + ...

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

- equational theory based on MLTT with intensional id.-types
- value terms are typed using a judgment $\Gamma \vdash_{\!v} V : A$

Computation terms: dep.-typed version of CBPV/EEC c. terms

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

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

A computational dep.-typed language

Value terms: MLTT's terms + thunks + ...

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

- equational theory based on MLTT with intensional id.-types
- value terms are typed using a judgment $\Gamma \vdash_{\mathrm{v}} V : A$

Computation terms: dep.-typed version of CBPV/EEC c. 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: These val. and comp. terms alone do not suffice, as in EEC!

A computational dep.-typed language

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

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

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

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

Computation and homomorphism terms are typed using judgments

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

Note: Formal presentation has more type-annotations on terms

A computational dep.-typed language

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

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

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

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

Computation and homomorphism terms are typed using judgments

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

Note: Formal presentation has more type-annotations on terms

A computational dep.-typed language

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

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

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

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

Computation and homomorphism terms are typed using judgments

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

Note: Formal presentation has more type-annotations on terms

A computational dep.-typed language

Typing rules: Dep.-typed versions of CBPV and EEC, e.g.:

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

$$\ldots$$

$$\frac{\Gamma \vdash \underline{C}}{\Gamma \,|\, z\!:\!\underline{C} \vDash_{\mathsf{h}} z : \underline{C}}$$

$$\ldots$$

$$\frac{\Gamma \vDash_{\mathsf{v}} V : A \qquad \Gamma \,|\, z\!:\!\underline{C} \vDash_{\mathsf{h}} K : \underline{D}[V/x]}{\Gamma \,|\, z\!:\!\underline{C} \vDash_{\mathsf{h}} \langle V, K \rangle : \Sigma x\!:\!A.\underline{D}}$$

$$\frac{\Gamma \,|\, z_1\!:\!\underline{C} \vDash_{\mathsf{h}} K : \Sigma x\!:\!A.\underline{D}_1 \qquad \Gamma \vdash \underline{D}_2 \qquad \Gamma, x\!:\!A \,|\, z_2\!:\!\underline{D}_1 \vDash_{\mathsf{h}} L : \underline{D}_2}{\Gamma \,|\, z_1\!:\!\underline{C} \vDash_{\mathsf{h}} K \;\mathtt{to}\; \langle x\!:\!A, z_2\!:\!\underline{D}_1 \rangle \;\mathtt{in}\; L : \underline{D}_2}$$

A computational dep.-typed language

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

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

The seq. comp. rule for λ_{ML} is justified by the type isomorphism

$$\Gamma \vdash \Sigma x{:}A.F(B) \cong F(\Sigma x{:}A.B)$$

Operations and equations

(primitives for programming with side-effects)

Algebraic operations and equations

Effect theories:

- we consider signatures of typed operation symbols

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

- equipped with equations on derivable effect terms
- type-dependency in operation symbols mostly a convenience

Algebraic operations:

$$\frac{\Gamma \models_{v} V : I \qquad \Gamma \vdash \underline{C} \qquad \Gamma, x : O[V/x_i] \models_{c} M : \underline{C}}{\Gamma \models_{c} \mathsf{op}^{\underline{C}}_{V}(x.M) : \underline{C}}$$

Generic effects:

$$\frac{\Gamma \models_{v} V : I}{\Gamma \models_{c} \mathtt{genop}_{V} : F\,(O[V/x_i])}$$

Example: Global store with two locations (modeled as booleans)

$$\mathsf{lookup} : (x_i : \mathsf{Bool}) \longrightarrow (\mathtt{if}\ x_i\ \mathtt{then}\ \mathsf{String}\ \mathtt{else}\ \mathsf{Nat})$$

$$\mathsf{update} : (x_i : \Sigma x : \mathsf{Bool}.(\mathtt{if}\ x\ \mathtt{then}\ \mathsf{String}\ \mathtt{else}\ \mathsf{Nat})) \longrightarrow 1$$

Algebraic operations and equations

Effect theories:

- we consider signatures of typed operation symbols

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

- equipped with equations on derivable effect terms
- type-dependency in operation symbols mostly a convenience

Algebraic operations:

$$\frac{\Gamma \models_{\mathsf{v}} V : I \quad \Gamma \vdash \underline{C} \quad \Gamma, x : O[V/x_i] \models_{\mathsf{c}} M : \underline{C}}{\Gamma \models_{\mathsf{c}} \mathsf{op}^{\underline{C}}_{V}(x.M) : \underline{C}}$$

Generic effects:

$$\frac{\Gamma \models_{\mathsf{v}} V : I}{\Gamma \models_{\mathsf{c}} \mathtt{genop}_V : F\,(O[V/x_i])}$$

Example: Global store with two locations (modeled as booleans)

$$\mathsf{lookup} : (x_i : \mathsf{Bool}) \longrightarrow (\mathtt{if}\ x_i\ \mathtt{then}\ \mathsf{String}\ \mathtt{else}\ \mathsf{Nat})$$

$$\mathsf{update} : (x_i : \Sigma x : \mathsf{Bool}.(\mathtt{if}\ x\ \mathtt{then}\ \mathsf{String}\ \mathtt{else}\ \mathsf{Nat})) \longrightarrow 1$$

Algebraic operations and equations

Effect theories:

- we consider signatures of typed operation symbols

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

- equipped with equations on derivable effect terms
- type-dependency in operation symbols mostly a convenience

Algebraic operations:

$$\frac{\Gamma \vdash_{\mathsf{v}} V : I \quad \Gamma \vdash \underline{C} \quad \Gamma, x : O[V/x_i] \vdash_{\mathsf{c}} M : \underline{C}}{\Gamma \vdash_{\mathsf{c}} \mathsf{op}^{\underline{C}}_{V}(x.M) : \underline{C}}$$

Generic effects:

$$\frac{\Gamma \vdash_{\mathsf{v}} V : I}{\Gamma \vdash_{\mathsf{c}} \mathtt{genop}_{V} : F\,(O[V/x_i])}$$

Example: Global store with two locations (modeled as booleans)

$$\mathsf{lookup} : (x_i : \mathsf{Bool}) \longrightarrow (\mathtt{if}\ x_i\ \mathtt{then}\ \mathsf{String}\ \mathtt{else}\ \mathsf{Nat})$$

$$\mathsf{update} : \big(x_i : \Sigma x : \mathsf{Bool}.(\mathtt{if}\ x\ \mathtt{then}\ \mathsf{String}\ \mathtt{else}\ \mathsf{Nat})\big) \longrightarrow 1$$

What about handlers?

We ensure that K's behave like homomorphisms via the rule

$$\Gamma \,|\, z \!:\! \underline{C} \models_{\mathsf{h}} K : \underline{D} \quad\Longrightarrow\quad \Gamma \models_{\mathsf{c}} K[\mathsf{op}^{\underline{C}}_{V}(x.M)/z] = \mathsf{op}^{\underline{D}}_{V}(x.K[M/z]) : \underline{D}$$

Recall: Plotkin-Pretnar presentation of handlers is given by:

$$\Gamma \models_{\mathsf{c}} M \;\texttt{handled with}\; \{\mathsf{op}_x(y) \mapsto M_{\mathsf{op}}\}_{\mathsf{op}} \;\mathsf{to}\; x \!:\! A \;\texttt{in}\; M_{\mathsf{ret}} : \underline{C}$$

- semantically, $\{\mathsf{op}_x(y) \mapsto M_{\mathsf{op}}\}_{\mathsf{op}}$ defines an algebra on $U[\![\underline{C}]\!]$
- and M `handled` … is the unique homomorphism out of $F[\![A]\!]$

Note: We have homomorphisms in the language, namely, the K's

Q: So, could we simply add?

$$\Gamma \,|\, z \!:\! \underline{C} \models_{\mathsf{h}} K \;\texttt{handled with}\; \{\mathsf{op}_x(y) \mapsto M_{\mathsf{op}}\}_{\mathsf{op}} \;\mathsf{to}\; x \!:\! A \;\texttt{in}\; M_{\mathsf{ret}} : \underline{D}$$

A: Unfortunately not — the algebra structure only at term level

What about handlers?

We ensure that K's behave like homomorphisms via the rule

$$\Gamma \mid z \mathbin{:} \underline{C} \vDash_{\mathsf{h}} K : \underline{D} \quad \Longrightarrow \quad \Gamma \vDash_{\mathsf{c}} K[\mathtt{op}^{\underline{C}}_{V}(x.M)/z] = \mathtt{op}^{\underline{D}}_{V}\big(x.K[M/z]\big) : \underline{D}$$

Recall: Plotkin-Pretnar presentation of handlers is given by:

$$\Gamma \vDash_{\mathsf{c}} M \ \mathtt{handled\ with}\ \{\mathsf{op}_x(y) \mapsto M_{\mathsf{op}}\}_{\mathsf{op}}\ \mathsf{to}\ x \mathbin{:} A\ \mathtt{in}\ M_{\mathsf{ret}} : \underline{C}$$

- semantically, $\{\mathsf{op}_x(y) \mapsto M_{\mathsf{op}}\}_{\mathsf{op}}$ defines an algebra on $U[\![\underline{C}]\!]$
- and M `handled` ... is the unique homomorphism out of $F[\![A]\!]$

Note: We have homomorphisms in the language, namely, the K's

Q: So, could we simply add?

$$\Gamma \mid z \mathbin{:} \underline{C} \vDash_{\mathsf{h}} K\ \mathtt{handled\ with}\ \{\mathsf{op}_x(y) \mapsto M_{\mathsf{op}}\}_{\mathsf{op}}\ \mathsf{to}\ x \mathbin{:} A\ \mathtt{in}\ M_{\mathsf{ret}} : \underline{D}$$

A: Unfortunately not — the algebra structure only at term level

What about handlers?

We ensure that K's behave like homomorphisms via the rule

$$\Gamma \mid z : \underline{C} \vdash_{\mathsf{h}} K : \underline{D} \quad\Longrightarrow\quad \Gamma \vdash_{\mathsf{c}} K[\mathsf{op}^{\underline{C}}_{V}(x.M)/z] = \mathsf{op}^{\underline{D}}_{V}\big(x.K[M/z]\big) : \underline{D}$$

Recall: Plotkin-Pretnar presentation of handlers is given by:

$$\Gamma \vdash_{\mathsf{c}} M \ \texttt{handled with}\ \{\mathsf{op}_x(y) \mapsto M_{\mathsf{op}}\}_{\mathsf{op}}\ \texttt{to}\ x : A\ \texttt{in}\ M_{\mathsf{ret}} : \underline{C}$$

- semantically, $\{\mathsf{op}_x(y) \mapsto M_{\mathsf{op}}\}_{\mathsf{op}}$ defines an algebra on $U[\![\underline{C}]\!]$
- and M `handled` $\ldots$ is the unique homomorphism out of $F[\![A]\!]$

Note: We have homomorphisms in the language, namely, the K's

Q: So, could we simply add?

$$\Gamma \mid z : \underline{C} \vdash_{\mathsf{h}} K \ \texttt{handled with}\ \{\mathsf{op}_x(y) \mapsto M_{\mathsf{op}}\}_{\mathsf{op}}\ \texttt{to}\ x : A\ \texttt{in}\ M_{\mathsf{ret}} : \underline{D}$$

A: Unfortunately not — the algebra structure only at term level

What about handlers?

We ensure that K's behave like homomorphisms via the rule

$$\Gamma \mid z : \underline{C} \vDash_{\mathsf{h}} K : \underline{D} \quad \Longrightarrow \quad \Gamma \vDash_{\mathsf{c}} K[\mathsf{op}^{\underline{C}}_{V}(x.M)/z] = \mathsf{op}^{\underline{D}}_{V}(x.K[M/z]) : \underline{D}$$

Recall: Plotkin-Pretnar presentation of handlers is given by:

$$\Gamma \vDash_{\mathsf{c}} M \ \mathtt{handled\ with}\ \{\mathsf{op}_x(y) \mapsto M_{\mathsf{op}}\}_{\mathsf{op}} \ \text{to}\ x : A \ \mathtt{in}\ M_{\mathsf{ret}} : \underline{C}$$

- semantically, $\{\mathsf{op}_x(y) \mapsto M_{\mathsf{op}}\}_{\mathsf{op}}$ defines an algebra on $U[\![\underline{C}]\!]$
- and M `handled` ... is the unique homomorphism out of $F[\![A]\!]$

Note: We have homomorphisms in the language, namely, the K's

Q: So, could we simply add?

$$\Gamma \mid z : \underline{C} \vDash_{\mathsf{h}} K \ \mathtt{handled\ with}\ \{\mathsf{op}_x(y) \mapsto M_{\mathsf{op}}\}_{\mathsf{op}} \ \text{to}\ x : A \ \mathtt{in}\ M_{\mathsf{ret}} : \underline{D}$$

A: Unfortunately not — the algebra structure only at term level

What about handlers?

We ensure that K's behave like homomorphisms via the rule

$$\Gamma \mid z : \underline{C} \vDash_{\mathsf{h}} K : \underline{D} \quad \Longrightarrow \quad \Gamma \vDash_{\mathsf{c}} K[\mathsf{op}^{\underline{C}}_{V}(x.M)/z] = \mathsf{op}^{\underline{D}}_{V}(x.K[M/z]) : \underline{D}$$

Recall: Plotkin-Pretnar presentation of handlers is given by:

$$\Gamma \vDash_{\mathsf{c}} M\ \texttt{handled with}\ \{\mathsf{op}_x(y) \mapsto M_{\mathsf{op}}\}_{\mathsf{op}}\ \mathsf{to}\ x : A\ \texttt{in}\ M_{\mathsf{ret}} : \underline{C}$$

- semantically, $\{\mathsf{op}_x(y) \mapsto M_{\mathsf{op}}\}_{\mathsf{op}}$ defines an algebra on $U[\![\underline{C}]\!]$
- and M `handled` ... is the unique homomorphism out of $F[\![A]\!]$

Note: We have homomorphisms in the language, namely, the K's

Q: So, could we simply add?

$$\Gamma \mid z : \underline{C} \vDash_{\mathsf{h}} K\ \texttt{handled with}\ \{\mathsf{op}_x(y) \mapsto M_{\mathsf{op}}\}_{\mathsf{op}}\ \mathsf{to}\ x : A\ \texttt{in}\ M_{\mathsf{ret}} : \underline{D}$$

A: Unfortunately not — the algebra structure only at term level

What about handlers?

We ensure that K's behave like homomorphisms via the rule

$$\Gamma \mid z\!:\!\underline{C} \vDash_{\mathsf{h}} K : \underline{D} \quad\Longrightarrow\quad \Gamma \vDash_{\mathsf{c}} K[\mathsf{op}^{\underline{C}}_{V}(x.M)/z] = \mathsf{op}^{\underline{D}}_{V}(x.K[M/z]) : \underline{D}$$

Recall: Plotkin-Pretnar presentation of handlers is given by:

$$\Gamma \vDash_{\mathsf{c}} M \;\texttt{handled with}\; \{\mathsf{op}_x(y) \mapsto M_{\mathsf{op}}\}_{\mathsf{op}} \;\texttt{to}\; x\!:\!A \;\texttt{in}\; M_{\mathsf{ret}} : \underline{C}$$

- semantically, $\{\mathsf{op}_x(y) \mapsto M_{\mathsf{op}}\}_{\mathsf{op}}$ defines an algebra on $U[\![\underline{C}]\!]$
- and M `handled` ... is the unique homomorphism out of $F[\![A]\!]$

Note: We have homomorphisms in the language, namely, the K's

Q: So, could we simply add?

$$\Gamma \mid z\!:\!\underline{C} \vDash_{\mathsf{h}} K \;\texttt{handled with}\; \{\mathsf{op}_x(y) \mapsto M_{\mathsf{op}}\}_{\mathsf{op}} \;\texttt{to}\; x\!:\!A \;\texttt{in}\; M_{\mathsf{ret}} : \underline{D}$$

A: Unfortunately not — the algebra structure only at term level

One way forward with handlers

User-defined algebra types:

(definitional equational proof obligations about V_{op}'s omitted)

$$\frac{\Gamma \vdash A \qquad \{\Gamma, x_1 : I, x_2 : O[x_1/x_i] \to A \vDash_{\mathsf{v}} V_{\mathsf{op}} : A\}_{\mathsf{op}:(x_i:I)\longrightarrow O}}{\Gamma \vdash \langle A, \{(x_1, x_2).V_{\mathsf{op}}\}_{\mathsf{op}:(x_i:I)\longrightarrow O}\rangle}$$

Introduction: $\mathtt{force}_{\langle A,\{(x_1,x_2).V_{\mathsf{op}}\}_{\mathsf{op}}\rangle}\, V$ (where $V : A$)

Elimination: (comp. term version)

(definitional equational proof obligations about N omitted)

$$\frac{\Gamma \vDash_{\mathsf{c}} M : \langle A, \{(x_1, x_2).V_{\mathsf{op}}\}_{\mathsf{op}}\rangle \qquad \Gamma, x : A \vDash_{\mathsf{c}} N : \underline{C}}{\Gamma \vDash_{\mathsf{c}} \mathtt{run}\ M\ \mathtt{as}\ x\ \mathtt{in}\ N : \underline{C}}$$

Equations:

- $U\langle A, \{(x, y).V_{\mathsf{op}}\}_{\mathsf{op}}\rangle = A$
- $\mathtt{op}_V^{\langle A,\{(x_1,x_2).V_{\mathsf{op}}\}_{\mathsf{op}}\rangle}(x.M) = \mathtt{force}\,(V_{\mathsf{op}}[V/x_1, \lambda x.\mathtt{thunk}\, M/x_2])$
- (η- and β-equations for intro.-elim. interaction)

One way forward with handlers

User-defined algebra types:

(definitional equational proof obligations about V_{op}'s omitted)

$$\frac{\Gamma \vdash A \qquad \{\Gamma, x_1 : I, x_2 : O[x_1/x_i] \to A \vDash_{\mathsf{v}} V_{\mathsf{op}} : A\}_{\mathsf{op}:(x_i:I) \longrightarrow O}}{\Gamma \vdash \langle A, \{(x_1, x_2).V_{\mathsf{op}}\}_{\mathsf{op}:(x_i:I) \longrightarrow O}\rangle}$$

Introduction: $\texttt{force}_{\langle A, \{(x_1,x_2).V_{\mathsf{op}}\}_{\mathsf{op}}\rangle} V$ (where $V : A$)

Elimination: (comp. term version)

(definitional equational proof obligations about N omitted)

$$\frac{\Gamma \vDash_{\mathsf{c}} M : \langle A, \{(x_1, x_2).V_{\mathsf{op}}\}_{\mathsf{op}}\rangle \qquad \Gamma, x : A \vDash_{\mathsf{c}} N : \underline{C}}{\Gamma \vDash_{\mathsf{c}} \texttt{run}\ M\ \texttt{as}\ x\ \texttt{in}\ N : \underline{C}}$$

Equations:

- $U\langle A, \{(x, y).V_{\mathsf{op}}\}_{\mathsf{op}}\rangle = A$
- $\texttt{op}_V^{\langle A, \{(x_1,x_2).V_{\mathsf{op}}\}_{\mathsf{op}}\rangle}(x.M) = \texttt{force}\,(V_{\mathsf{op}}[V/x_1, \lambda x.\texttt{thunk}\,M/x_2])$
- (η- and β-equations for intro.-elim. interaction)

One way forward with handlers

User-defined algebra types:

(definitional equational proof obligations about V_{op}'s omitted)

$$\frac{\Gamma \vdash A \qquad \{\Gamma, x_1 : I, x_2 : O[x_1/x_i] \to A \vDash_{\mathsf{v}} V_{\mathsf{op}} : A\}_{\mathsf{op}:(x_i:I)\longrightarrow O}}{\Gamma \vdash \langle A, \{(x_1, x_2).V_{\mathsf{op}}\}_{\mathsf{op}:(x_i:I)\longrightarrow O}\rangle}$$

Introduction: $\mathtt{force}_{\langle A,\{(x_1,x_2).V_{\mathsf{op}}\}_{\mathsf{op}}\rangle} V$ (where $V : A$)

Elimination: (comp. term version)

(definitional equational proof obligations about N omitted)

$$\frac{\Gamma \vDash_{\mathsf{c}} M : \langle A, \{(x_1, x_2).V_{\mathsf{op}}\}_{\mathsf{op}}\rangle \qquad \Gamma, x{:}A \vDash_{\mathsf{c}} N : \underline{C}}{\Gamma \vDash_{\mathsf{c}} \mathtt{run}\ M\ \mathtt{as}\ x\ \mathtt{in}\ N : \underline{C}}$$

Equations:

- $U\langle A, \{(x, y).V_{\mathsf{op}}\}_{\mathsf{op}}\rangle = A$
- $\mathtt{op}_V^{\langle A,\{(x_1,x_2).V_{\mathsf{op}}\}_{\mathsf{op}}\rangle}(x.M) = \mathtt{force}\,(V_{\mathsf{op}}[V/x_1, \lambda x.\mathtt{thunk}\,M/x_2])$
- (η- and β-equations for intro.-elim. interaction)

One way forward with handlers

User-defined algebra types:

(definitional equational proof obligations about V_{op}'s omitted)

$$\frac{\Gamma \vdash A \qquad \{\Gamma, x_1 : I, x_2 : O[x_1/x_i] \to A \vdash_{\mathsf{v}} V_{\mathsf{op}} : A\}_{\mathsf{op}:(x_i:I)\longrightarrow O}}{\Gamma \vdash \langle A, \{(x_1, x_2).V_{\mathsf{op}}\}_{\mathsf{op}:(x_i:I)\longrightarrow O}\rangle}$$

Introduction: $\mathtt{force}_{\langle A,\{(x_1,x_2).V_{\mathsf{op}}\}_{\mathsf{op}}\rangle}\, V$ (where $V : A$)

Elimination: (comp. term version)

(definitional equational proof obligations about N omitted)

$$\frac{\Gamma \vdash_{\mathsf{c}} M : \langle A, \{(x_1, x_2).V_{\mathsf{op}}\}_{\mathsf{op}}\rangle \qquad \Gamma, x : A \vdash_{\mathsf{c}} N : \underline{C}}{\Gamma \vdash_{\mathsf{c}} \mathtt{run}\ M\ \mathtt{as}\ x\ \mathtt{in}\ N : \underline{C}}$$

Equations:

- $U\langle A, \{(x, y).V_{\mathsf{op}}\}_{\mathsf{op}}\rangle = A$
- $\mathtt{op}_V^{\langle A,\{(x_1,x_2).V_{\mathsf{op}}\}_{\mathsf{op}}\rangle}(x.M) = \mathtt{force}\,(V_{\mathsf{op}}[V/x_1, \lambda x.\mathtt{thunk}\, M/x_2])$
- (η- and β-equations for intro.-elim. interaction)

One way forward with handlers

User-defined algebra type:

(equational proof obligations about V_{op}'s omitted)

$$\frac{\Gamma \vdash A \qquad \{\Gamma, x : I, y : O[x/xi] \to A \vdash_{\mathsf{v}} V_{\mathrm{op}} : A\}_{\mathrm{op}:(x_i:I)\longrightarrow O}}{\Gamma \vdash \langle A, \{(x,y).V_{\mathrm{op}}\}_{\mathrm{op}:(x_i:I)\longrightarrow O}\rangle}$$

Encoding Plotkin-Pretnar handlers:

$$M\ \mathtt{handled\ with}\ \{\mathrm{op}_x(y) \mapsto M_{\mathrm{op}}\}_{\mathrm{op}}\ \mathtt{to}\ x{:}A\ \mathtt{in}\ M_{\mathrm{ret}} : \underline{C}$$

$$\overset{\text{def}}{=}$$

$$\mathtt{force}_{\underline{C}}\left(\underbrace{\mathtt{thunk}\left(M\ \mathtt{to}\ x{:}A\ \mathtt{in}\ \underbrace{\mathtt{force}_{\langle U\underline{C},\ldots\mathtt{thunk}\,(M_{\mathrm{op}})\ldots\rangle}\left(\underbrace{\mathtt{thunk}\ M_{\mathrm{ret}}}_{:U\underline{C}}\right)}_{:\langle U\underline{C},\ldots\mathtt{thunk}\,(M_{\mathrm{op}})\ldots\rangle}\right)}_{:U\underline{C}}\right)$$

Categorical semantics

(fibrations and adjunctions)

Categorical semantics

Using fibred cat. theory, we define fibred adjunction models

- a sound and complete class of models

given by: i) a split closed comprehension category $\mathcal{P}$

$$\begin{tikzcd}
\mathcal{B}^{\to} \arrow[dr, bend right, "\mathrm{cod}"'] & & \mathcal{V} \arrow[ll, "\mathcal{P}"'] \arrow[d, bend right=60, "p"'] \arrow[d, bend left=60, "\{-\}"] \\
 & & \mathcal{B} \arrow[u, "\dashv\ 1\ \dashv" description]
\end{tikzcd}$$

- following Streicher and Hoffmann, we have a partial interpretation function $[\![-]\!]$ on raw syntax, that maps (if defined):
- a context Γ to and object $[\![\Gamma]\!]$ in $\mathcal{B}$
- 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 $[\![\Gamma; V]\!] : 1_{[\![\Gamma]\!]} \to X$ in $\mathcal{V}_{[\![\Gamma]\!]}$

Categorical semantics

Using fibred cat. theory, we define fibred adjunction models

- a sound and complete class of models

given by: i) a split closed comprehension category $\mathcal{P}$

$$\begin{tikzcd}
\mathcal{B}^{\to} \arrow[ddr, bend right, "\mathrm{cod}"'] & & \mathcal{V} \arrow[ll, "\mathcal{P}"'] \arrow[dd, bend right=60, "p"'] \arrow[dd, bend left=60, "\{-\}"] \\
 & & \\
 & & \mathcal{B} \arrow[uu, "1"'] \arrow[uu, phantom, shift left=3ex, "\dashv"] \arrow[uu, phantom, shift right=3ex, "\dashv"]
\end{tikzcd}$$

- following Streicher and Hoffmann, we have a partial interpretation function $[\![-]\!]$ on raw syntax, that maps (if defined):
- a context Γ to and object $[\![\Gamma]\!]$ in $\mathcal{B}$
- 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 $[\![\Gamma; V]\!] : 1_{[\![\Gamma]\!]} \to X$ in $\mathcal{V}_{[\![\Gamma]\!]}$

Categorical semantics

Using fibred cat. theory, we define fibred adjunction models

- a sound and complete class of models

given by: i) a split closed comprehension category $\mathcal{P}$

$$\begin{tikzcd}
\mathcal{B}^{\to} \arrow[dr, bend right, "\mathrm{cod}"'] & \mathcal{V} \arrow[l, "\mathcal{P}"'] \arrow[d, bend right=60, "p"'] \arrow[d, bend left=60, "\{-\}"] \\
& \mathcal{B} \arrow[u, "\dashv", "1 \dashv"']
\end{tikzcd}$$

- the display maps $\pi_A = \mathcal{P}(A) : \{A\} \longrightarrow p(A)$ in $\mathcal{B}$
- induce the weakening functors $\pi_A^* : \mathcal{V}_{p(A)} \longrightarrow \mathcal{V}_{\{A\}}$
- and the value Σ- and Π-types are interpreted as adjoints

$$\Sigma_A \dashv \pi_A^* \dashv \Pi_A$$

(Σ_A is also required to be strong, i.e., support dep. elimination)

Categorical semantics

Using fibred cat. theory, we define fibred adjunction models

- a sound and complete class of models

given by: ii) a split fibration q and a split fib. adj. $F \dashv U$

$$\begin{tikzcd}
\mathcal{B}^{\to} \arrow[dr, bend right, "\mathrm{cod}"'] & \mathcal{V} \arrow[l, "\mathcal{P}"'] \arrow[r, bend left, "F"] \arrow[d, bend right=60, "p"'] \arrow[d, bend left=60, "\{-\}"] & \mathcal{C} \arrow[l, bend left, "U"] \arrow[l, phantom, "\perp"] \arrow[dl, bend left, "q"] \\
& \mathcal{B} \arrow[u, "\dashv", "1 \dashv"'] &
\end{tikzcd}$$

- we extend $[\![-]\!]$ so that it maps (if defined):
- 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 $[\![\Gamma; M]\!] : 1_{[\![\Gamma]\!]} \to U(Z)$ in $\mathcal{V}_{[\![\Gamma]\!]}$
- a ctx. Γ, a comp. type $\underline{C}$ and a hom. term K to
  $[\![\Gamma; \underline{C}; K]\!] : [\![\Gamma; \underline{C}]\!] \to Z$ in $\mathcal{C}_{[\![\Gamma]\!]}$

Categorical semantics

Using fibred cat. theory, we define fibred adjunction models

- a sound and complete class of models

given by: ii) a split fibration q and a split fib. adj. $F \dashv U$

$$\begin{tikzcd}
\mathcal{B}^{\to} \arrow[dr, bend right, "\mathrm{cod}"'] & \mathcal{V} \arrow[l, "\mathcal{P}"'] \arrow[r, bend left, "F"] \arrow[r, phantom, "\perp"] \arrow[d, bend right=60, "p"'] \arrow[d, bend left=60, "\{-\}"] & \mathcal{C} \arrow[l, bend left, "U"] \arrow[dl, bend left, "q"] \\
 & \mathcal{B} \arrow[u, "\dashv", "1 \dashv"'] &
\end{tikzcd}$$

- the display maps $\pi_A = \mathcal{P}(A) : \{A\} \longrightarrow p(A)$ in $\mathcal{B}$
- induce the weakening functors $\pi_A^* : \mathcal{C}_{p(A)} \longrightarrow \mathcal{C}_{\{A\}}$
- and the comp. Σ- and Π-types are interpreted again as adjoints

$$\Sigma_A \dashv \pi_A^* \dashv \Pi_A$$

Examples of fibred adjunction models

- for a split closed comprehension cat. $\mathcal{P} : \mathcal{V} \longrightarrow \mathcal{B}^{\rightarrow}$, we have
$$\mathsf{Id}_{\mathcal{V}} \dashv \mathsf{Id}_{\mathcal{V}} : \mathcal{V} \longrightarrow \mathcal{V}$$
- for a model of EEC ($\mathcal{V}$ is CCC, $\mathcal{C}$ is $\mathcal{V}$-enriched, $\mathcal{V}$-enr. adj., etc.)
$$F_{\mathsf{EEC}} \dashv U_{\mathsf{EEC}} : \mathsf{s}(\mathcal{V}, \mathcal{C}) \longrightarrow \mathsf{s}(\mathcal{V})$$
- for $\mathcal{P}_{\mathsf{fam}} : \mathsf{Fam}(\mathsf{Set}) \longrightarrow \mathsf{Set}^{\rightarrow}$ and $F \dashv U : \mathcal{C} \longrightarrow \mathsf{Set}$, when $\mathcal{C}$ has set-indexed products and set-indexed coproducts, we have
$$\widehat{F} \dashv \widehat{U} : \mathsf{Fam}(\mathcal{C}) \longrightarrow \mathsf{Fam}(\mathsf{Set})$$
- for any monad $T : \mathsf{Set} \longrightarrow \mathsf{Set}$ and $\mathcal{P}_{\mathsf{fam}} : \mathsf{Fam}(\mathsf{Set}) \longrightarrow \mathsf{Set}^{\rightarrow}$
$$\widehat{F^T} \dashv \widehat{U^T} : \mathsf{Fam}(\mathsf{Set}^T) \longrightarrow \mathsf{Fam}(\mathsf{Set})$$
- for the continuations monad $R^{R^{(-)}} : \mathsf{Set} \longrightarrow \mathsf{Set}$, we have
$$\widehat{R^{(-)}} \dashv \widehat{R^{(-)}} : \mathsf{Fam}(\mathsf{Set}^{op}) \longrightarrow \mathsf{Fam}(\mathsf{Set})$$
and analogously for the state monad $(S \times (-))^S$

Examples of fibred adjunction models

- for a split closed comprehension cat. $\mathcal{P} : \mathcal{V} \longrightarrow \mathcal{B}^{\rightarrow}$, we have
$$\mathsf{Id}_{\mathcal{V}} \dashv \mathsf{Id}_{\mathcal{V}} : \mathcal{V} \longrightarrow \mathcal{V}$$
- for a model of EEC ($\mathcal{V}$ is CCC, $\mathcal{C}$ is $\mathcal{V}$-enriched, $\mathcal{V}$-enr. adj., etc.)
$$F_{\mathsf{EEC}} \dashv U_{\mathsf{EEC}} : \mathsf{s}(\mathcal{V}, \mathcal{C}) \longrightarrow \mathsf{s}(\mathcal{V})$$
- for $\mathcal{P}_{\mathsf{fam}} : \mathsf{Fam}(\mathsf{Set}) \longrightarrow \mathsf{Set}^{\rightarrow}$ and $F \dashv U : \mathcal{C} \longrightarrow \mathsf{Set}$, when $\mathcal{C}$ has set-indexed products and set-indexed coproducts, we have
$$\widehat{F} \dashv \widehat{U} : \mathsf{Fam}(\mathcal{C}) \longrightarrow \mathsf{Fam}(\mathsf{Set})$$
- for any monad $T : \mathsf{Set} \longrightarrow \mathsf{Set}$ and $\mathcal{P}_{\mathsf{fam}} : \mathsf{Fam}(\mathsf{Set}) \longrightarrow \mathsf{Set}^{\rightarrow}$
$$\widehat{F^T} \dashv \widehat{U^T} : \mathsf{Fam}(\mathsf{Set}^T) \longrightarrow \mathsf{Fam}(\mathsf{Set})$$
- for the continuations monad $R^{R^{(-)}} : \mathsf{Set} \longrightarrow \mathsf{Set}$, we have
$$\widehat{R^{(-)}} \dashv \widehat{R^{(-)}} : \mathsf{Fam}(\mathsf{Set}^{op}) \longrightarrow \mathsf{Fam}(\mathsf{Set})$$
and analogously for the state monad $(S \times (-))^S$

Examples of fibred adjunction models

- for a split closed comprehension cat. $\mathcal{P} : \mathcal{V} \longrightarrow \mathcal{B}^{\to}$, we have

$$\mathsf{Id}_{\mathcal{V}} \dashv \mathsf{Id}_{\mathcal{V}} : \mathcal{V} \longrightarrow \mathcal{V}$$

- for a model of EEC ($\mathcal{V}$ is CCC, $\mathcal{C}$ is $\mathcal{V}$-enriched, $\mathcal{V}$-enr. adj., etc.)

$$F_{\mathsf{EEC}} \dashv U_{\mathsf{EEC}} : \mathsf{s}(\mathcal{V}, \mathcal{C}) \longrightarrow \mathsf{s}(\mathcal{V})$$

- for $\mathcal{P}_{\mathsf{fam}} : \mathsf{Fam}(\mathsf{Set}) \longrightarrow \mathsf{Set}^{\to}$ and $F \dashv U : \mathcal{C} \longrightarrow \mathsf{Set}$, when $\mathcal{C}$ has set-indexed products and set-indexed coproducts, we have

$$\widehat{F} \dashv \widehat{U} : \mathsf{Fam}(\mathcal{C}) \longrightarrow \mathsf{Fam}(\mathsf{Set})$$

- for any monad $T : \mathsf{Set} \longrightarrow \mathsf{Set}$ and $\mathcal{P}_{\mathsf{fam}} : \mathsf{Fam}(\mathsf{Set}) \longrightarrow \mathsf{Set}^{\to}$

$$\widehat{F^T} \dashv \widehat{U^T} : \mathsf{Fam}(\mathsf{Set}^T) \longrightarrow \mathsf{Fam}(\mathsf{Set})$$

- for the continuations monad $R^{R^{(-)}} : \mathsf{Set} \longrightarrow \mathsf{Set}$, we have

$$\widehat{R^{(-)}} \dashv \widehat{R^{(-)}} : \mathsf{Fam}(\mathsf{Set}^{\mathsf{op}}) \longrightarrow \mathsf{Fam}(\mathsf{Set})$$

and analogously for the state monad $(S \times (-))^S$

Examples of fibred adjunction models

- for a split closed comprehension cat. $\mathcal{P} : \mathcal{V} \longrightarrow \mathcal{B}^{\rightarrow}$, we have

$$\mathsf{Id}_{\mathcal{V}} \dashv \mathsf{Id}_{\mathcal{V}} : \mathcal{V} \longrightarrow \mathcal{V}$$

- for a model of EEC ($\mathcal{V}$ is CCC, $\mathcal{C}$ is $\mathcal{V}$-enriched, $\mathcal{V}$-enr. adj., etc.)

$$F_{\mathsf{EEC}} \dashv U_{\mathsf{EEC}} : \mathsf{s}(\mathcal{V}, \mathcal{C}) \longrightarrow \mathsf{s}(\mathcal{V})$$

- for $\mathcal{P}_{\mathsf{fam}} : \mathsf{Fam}(\mathsf{Set}) \longrightarrow \mathsf{Set}^{\rightarrow}$ and $F \dashv U : \mathcal{C} \longrightarrow \mathsf{Set}$, when $\mathcal{C}$ has set-indexed products and set-indexed coproducts, we have

$$\widehat{F} \dashv \widehat{U} : \mathsf{Fam}(\mathcal{C}) \longrightarrow \mathsf{Fam}(\mathsf{Set})$$

- for any monad $T : \mathsf{Set} \longrightarrow \mathsf{Set}$ and $\mathcal{P}_{\mathsf{fam}} : \mathsf{Fam}(\mathsf{Set}) \longrightarrow \mathsf{Set}^{\rightarrow}$

$$\widehat{F^T} \dashv \widehat{U^T} : \mathsf{Fam}(\mathsf{Set}^T) \longrightarrow \mathsf{Fam}(\mathsf{Set})$$

- for the continuations monad $R^{R^{(-)}} : \mathsf{Set} \longrightarrow \mathsf{Set}$, we have

$$\widehat{R^{(-)}} \dashv \widehat{R^{(-)}} : \mathsf{Fam}(\mathsf{Set}^{\mathsf{op}}) \longrightarrow \mathsf{Fam}(\mathsf{Set})$$

  and analogously for the state monad $(S \times (-))^S$

Examples of fibred adjunction models

Another example:

- for a $\mathcal{CPO}$-enriched monad $T : \mathcal{CPO} \longrightarrow \mathcal{CPO}$ with a least algebraic operation $\Omega : 0$ and reflexive coequalizers in $\mathcal{CPO}^T$

  $$\widehat{F^T} \dashv \widehat{U^T} : \mathsf{CFam}(\mathcal{CPO}^T) \longrightarrow \mathsf{CFam}(\mathcal{CPO})$$

  where $\mathsf{CFam}(\mathcal{CPO})$ is the cat. of continuous families

  $$\Big((X, \sqsubseteq_X)\,,\, A : (X, \sqsubseteq_X) \longrightarrow \mathcal{CPO}^{\mathsf{EP}}\Big)$$

- this allows us to treat general recursion as a comp. effect by

  $$\frac{\Gamma, x\!:\!U\underline{C} \vdash_{\mathsf{c}} M : \underline{C}}{\Gamma \vdash_{\mathsf{c}} \mu x\!:\!U\underline{C}.M : \underline{C}}$$

- but have to restrict A in $\mathsf{Id}_A(V, W)$ to be discrete to define

  $$\mathsf{Id}_{(X,A)} \overset{\mathsf{def}}{=} \big(\{\pi^*_{(X,A)}(X, A)\}, \langle x, a, a' \rangle \mapsto \textstyle\coprod_{\{\star \,|\, a = a'\}} 1\big)$$

Conclusions

A dependently-typed computational language with

- clear distinction between values and computations
- systematic treatment of seq. composition (comp. Σ-types)
- algebraic effects and handlers
- natural denotational semantics, using standard math. tools

Ongoing work

- integrating dependent- and effect-typing
  - e.g., fibred parametrised adjunctions for a principled account of resource-dependent effects in Idris

$$\texttt{EffM}\ \varepsilon_1 \Big((x : A).\varepsilon_2(x) \Big) \qquad = \qquad U_{\varepsilon_1} \Big(\Sigma x : A . F_{\varepsilon_2(x)}(1) \Big)$$

- homomorphic type-dependency on effectful computations

Thank you for listening!

Conclusions

A dependently-typed computational language with

- clear distinction between values and computations
- systematic treatment of seq. composition (comp. Σ-types)
- algebraic effects and handlers
- natural denotational semantics, using standard math. tools

Ongoing work

- integrating dependent- and effect-typing
  - e.g., fibred parametrised adjunctions for a principled account of resource-dependent effects in Idris

$$\texttt{EffM}\ \varepsilon_1 \Big((x : A) . \varepsilon_2(x) \Big) \qquad = \qquad U_{\varepsilon_1} \Big(\Sigma x : A . F_{\varepsilon_2(x)}(1) \Big)$$

- homomorphic type-dependency on effectful computations

Thank you for listening!