Modal Effect Types

Wenhao Tang

The University of Edinburgh

SPLV Lightning Talk, 30th July 2024

(Joint work with Leo White, Stephen Dolan, Daniel Hillerström, Sam Lindley, Anton Lorenzen)

Verbosity of Conventional Effect Types

```
map : ∀ a b . (a → b, List a) → List b
```

Verbosity of Conventional Effect Types

$$\texttt{map} : \forall\ \texttt{a}\ \texttt{b}\ .\ (\texttt{a} \to \texttt{b},\ \texttt{List}\ \texttt{a}) \to \texttt{List}\ \texttt{b}$$

$$\texttt{map} : \forall\ \texttt{a}\ \texttt{b}\ .\ (\texttt{a} \xrightarrow{\texttt{IO}} \texttt{b},\ \texttt{List}\ \texttt{a}) \xrightarrow{\texttt{IO}} \texttt{List}\ \texttt{b}$$

$$\texttt{map} : \forall\ \texttt{a}\ \texttt{b}\ .\ (\texttt{a} \xrightarrow{\texttt{Exception}} \texttt{b},\ \texttt{List}\ \texttt{a}) \xrightarrow{\texttt{Exception}} \texttt{List}\ \texttt{b}$$

$$\texttt{map} : \forall\ \texttt{a}\ \texttt{b}\ .\ (\texttt{a} \xrightarrow{\texttt{IO, Exception}} \texttt{b},\ \texttt{List}\ \texttt{a}) \xrightarrow{\texttt{IO, Exception}} \texttt{List}\ \texttt{b}$$

Verbosity of Conventional Effect Types

```
map : ∀ a b . (a → b, List a) → List b
```

```
map : ∀ a b . (a -IO-> b, List a) -IO-> List b
map : ∀ a b . (a -Exception-> b, List a) -Exception-> List b
map : ∀ a b . (a -IO, Exception-> b, List a) -IO, Exception-> List b
```

We need effect variables to apply `map` to arbitrarily effectful functions.

```
map : ∀ a b e . (a -e-> b, List a) -e-> List b
```

Effect Contexts

The core idea is to *decouple effects from function arrows*, and manage effects as ***effect contexts***.

Effect Contexts

The core idea is to *decouple effects from function arrows*, and manage effects as ***effect contexts***.

Functions can use any effects from the context by default.

Effect Contexts

The core idea is to *decouple effects from function arrows*, and manage effects as ***effect contexts***.

Functions can use any effects from the context by default.

```
map : ∀ a b . (a → b, List a) → List b
```

This `map` can be applied to any effectful functions. Both the parameter and result functions can use any effects from the context.

Effect Contexts

The core idea is to *decouple effects from function arrows*, and manage effects as ***effect contexts***.

Functions can use any effects from the context by default.

```
map : ∀ a b . (a → b, List a) → List b
```

This `map` can be applied to any effectful functions. Both the parameter and result functions can use any effects from the context.

No change is required for first-class higher-order functions.

```
map : ∀ a b . (a → b) → List a → List b
```

Absolute Modalities

An ***absolute modality*** specifies an effect context.

Absolute Modalities

An *absolute modality* specifies an effect context.

For `map`, the `[]` indicates that it does not require any effects.

```
map : ∀ a b . [](a → b, List a) → List b
```

Absolute Modalities

An *absolute modality* specifies an effect context.

For `map`, the `[]` indicates that it does not require any effects.

```
map : ∀ a b . [](a → b, List a) → List b
```

For a generator, the `[yield]` indicates that it might use `yield`.

```
gen : [yield](List Int → 1)
gen xs = map (fun x → do yield x) xs; ()
```

Absolute Modalities

An *absolute modality* specifies an effect context.

For `map`, the `[]` indicates that it does not require any effects.

```
map : ∀ a b . [](a → b, List a) → List b
```

For a generator, the `[yield]` indicates that it might use `yield`.

```
gen : [yield](List Int → 1)
gen xs = map (fun x → do yield x) xs; ()
```

With conventional effect types, we usually need an effect variable.

$$\texttt{gen} : \forall\ \texttt{e}\ .\ \texttt{List Int} \xrightarrow{\texttt{yield, e}} \texttt{1}$$

Relative Modalities

A *relative modality* specifies a local changes to an effect context.

Relative Modalities

A *relative modality* specifies a local changes to an effect context.

Useful to give modular types to effect handlers.

```
asList : <yield>(1 → 1) → List Int
asList m = handle m () with
  return ()  ⇒ nil
  yield  x r ⇒ cons x (r ())
```

Relative Modalities

A *relative modality* specifies a local changes to an effect context.

Useful to give modular types to effect handlers.

```
asList : <yield>(1 → 1) → List Int
asList m = handle m () with
  return ()  ⇒ nil
  yield  x r ⇒ cons x (r ())
```

The `<yield>` extends the effect context with the `yield` effect, which is handled by an effect handler in `asList`.

Relative Modalities

A *relative modality* specifies a local changes to an effect context.

Useful to give modular types to effect handlers.

```
asList : <yield>(1 → 1) → List Int
asList m = handle m () with
  return ()  ⇒ nil
  yield  x r ⇒ cons x (r ())
```

The `<yield>` extends the effect context with the `yield` effect, which is handled by an effect handler in `asList`.

A function `<yield>(1 → 1)` can still use any effects from the context.

Relative Modalities

A *relative modality* specifies a local changes to an effect context.

Useful to give modular types to effect handlers.

```
asList : <yield>(1 → 1) → List Int
asList m = handle m () with
  return ()  ⇒ nil
  yield  x r ⇒ cons x (r ())
```

The `<yield>` extends the effect context with the `yield` effect, which is handled by an effect handler in `asList`.

A function `<yield>(1 → 1)` can still use any effects from the context.

With conventional effect types, we usually need an effect variable.

$$\texttt{asList} : \forall\ \texttt{e}\ .\ (\texttt{1} \xrightarrow{\texttt{yield, e}} \texttt{1}) \xrightarrow{\texttt{e}} \texttt{List Int}$$

Modular Effectful Programming

```
> asList <yield>(fun () → gen [3,1,4,1,5,9])
# [3,1,4,1,5,9] : List Int
```

Modular Effectful Programming

```
> asList <yield>(fun () → gen [3,1,4,1,5,9])
# [3,1,4,1,5,9] : List Int
```

```
state : ∀ [a] . <get, put>(1 → a) → Int → (a, Int)
gen'  : [yield, get, put](List Int → 1)
```

```
> asList <yield>(fun () →
    state <get,put>(fun () → gen' [3,1,4,1,5,9]) 0; ())
# [3,4,8,9,14,23] : List Int
```

More in the Paper

Met: A core calculus following simple *multimodal type theory*.
Encoding a fragment of conventional effect types into Met

Mete: Extension with *effect variables*.

Metel: A surface language with *sound and complete type inference*.

https://arxiv.org/abs/2407.11816