

Intrinsically-Typed Interpreters for Effectful and Coeffectful Languages

Syouki Tsuyama Youyou Cong Hidehiko Masuhara
Tokyo Institute of Technology

Intrinsically-Typed Interpreter (ITI)

An executable, typed specification of semantics

$\text{eval} : \underline{\text{Expr } T} \rightarrow \text{Val } T$

Expression of type T evaluates to value T

= Statement of type safety


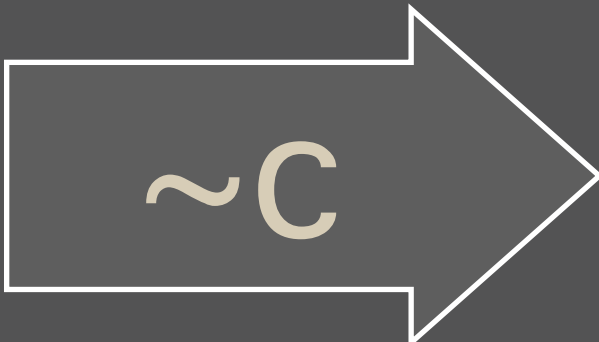
Our Research : ITI for Effects and Coeffects

- ✓ Delimited control operators [Danvi&Filinski1990]
- ✓ Algebraic effects and handlers [Pretnar2015]
- Asynchronous Effects [Ahman POPL'21]
- Trace Effects [Skalka JFP'18]
- Quantitative types [Atkey LICS'18]

ITI for Algebraic Effects & Handlers

Based on CPS translation for effect handlers^[Hillerstrom FSCD'17]

e.g. Interpretation of `return ()`

λe (object language)			Agda value
Term	<code>Return unit</code>	 <code>evalc</code>	$\lambda k \rightarrow \lambda h \rightarrow k \text{ tt}$
Type	<code>Unit!E</code>	 $\sim c$	$\forall \{T : \text{Set}\} \rightarrow$ $\text{Cont} \rightarrow \text{Hand} \rightarrow T$

Syntax of λe

Values $V, W ::= \text{unit} \mid \lambda x. M$

Computations $M, N ::= V W \mid \text{return } V \mid \text{do } \lambda V \mid$
 $\text{let } x = M \text{ in } N \mid$
 $\text{handle } M \text{ with } H$

Handlers $H ::= \{\text{return } x \rightarrow M\} \mid$
 $H \cup \{\lambda p k \rightarrow M\}$

Evaluation of λe

✓ `handle (let x = do choose() in not x)`
`with {`
 `return x -> return x`
 `choose _ k -> k true`
`}` (* result : false *)

✗ `let x = do choose() in not x`

Unhandled operation

Type of ITI for λe

Type-safety: Pure computation of type A is evaluated into value of type A

$\text{eval} : \text{Cmp } \sim t (A, []) \rightarrow \sim t A$

Organization of ITI for λe

- `eval` executes top-level programs.

```
eval e = evalc e init-k pure-h
```

- `evalc` , `evalv` (for computations and values)

```
evalc : Cmp ~t C -> ~c C  
evalv : Val ~t A -> ~t A
```


Evaluation of computations

`evalc` executes an effectful computation
with a continuation and a handler

continuation

handler

```
evalc (Do label v) k h  
= h label (evalv v) ( $\lambda x \rightarrow k\ x\ h$ )
```

Evaluation of Values

`evalv` translates values into Agda values

```
evalv unit      = tt
evalv (var x)    = x
evalv (fun f)    =  $\lambda x \rightarrow \text{evalc } (f\ x)$ 
```

Running Example



```
-- return ()  
test0 : eval (Return unit) ≡ tt  
test0 = refl
```

```
-- let x = do choose() in (not x)  
e : Cmp ~t (Bool , Choose)  
e = Let (Do (here refl) unit) In (λ x -> not · (var x))
```



```
testT : eval (Handle e With ChooseTrue) ≡ false  
testT = refl
```



```
eval e    ➡ type error!
```

Future Work

Develop ITI for other features

- Asynchronous Effects[Ahman POPL'21]
- Trace Effects[Skalka JFP'18]
- Quantitative types[Atkey LICS'18]

➡ Additional proof terms complicate interpreters

Hide them using abstraction

ITI for Mutable State [Poulsen POPL'18]

$\text{eval} : \mathbb{N} \rightarrow \forall \{\Sigma \ \Gamma \ t\} \rightarrow \text{Expr} \ \Gamma \ t \rightarrow \textcolor{blue}{M} \ \Gamma \ (\text{Val } t) \ \Sigma$

Abstraction

$\textcolor{blue}{M} \ \Gamma \ p \ \Sigma = \text{Env} \ \Gamma \ \Sigma \rightarrow \text{Store} \ \Sigma \rightarrow$
 $\text{Maybe} \ (\exists \lambda \ \underline{\Sigma'} \rightarrow \text{Store} \ \Sigma' \times p \ \Sigma' \times \underline{\Sigma \sqsubseteq \Sigma'})$

Store keeps growing
throughout evaluation

Discussion Topics

- Interesting features to consider
- Possible abstractions for different features

ITI with Advanced features

Features of existing ITI

Additional proof

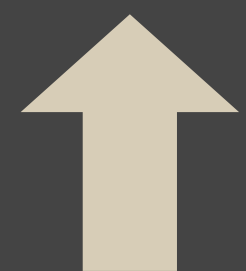
Mutable state[Poulsen POPL'18]

Type-safe read&write

Linear types[Rouvoet CPP'20]

Linear usage of variables

Additional proof terms complicate interpreters



They hide proof terms using abstraction