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

```
eval : Expr T → 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	`Return unit`	evalc	`λk → λh → k tt`
Type	`Unit!E`	~c	`∀{T : Set} → Cont → Hand → T`

Syntax of λe

Values V, W ::= unit | λx.M

Computations M, N ::= V W | return V | do l V | let x = M in N | handle M with H

Handlers H ::= {return x -> M} | H ⊎ {l p k -> M}

```
Values        V, W ::= unit | λx.M

Computations  M, N ::= V W | return V | do l V |
                       let x = M in N |
                       handle M with H

Handlers      H    ::= {return x -> M} |
                       H ⊎ {l p k -> 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

```
eval : Cmp ~t (A , []) -> ~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) (λ x -> k x h)
```

Evaluation of Values

`evalv` translates values into Agda values

```
evalv unit      = tt
evalv (var x)   = x
evalv (fun f)   = λ x -> 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]

```
eval : ℕ → ∀ {Σ Γ t} → Expr Γ t → M Γ (Val t) Σ
```

Abstraction

```
M Γ p Σ = Env Γ Σ → Store Σ →
          Maybe (∃ λ Σ' → Store Σ' × p Σ' × Σ ⊑ Σ')
```

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