

# 副作用を持つプログラミング言語を 型安全に定義するためのライブラリの開発に向けて

東京工業大学 数理・計算科学系 増原研究室

18B10155 津山勝輝

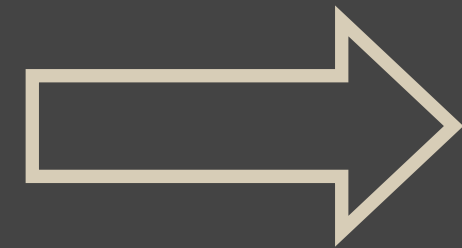
2022年2月10日 卒業論文発表

# 背景：型付き言語の実装

対象言語

メタ言語  
(実装に使う言語)

型システム



型検査器

意味論



インタプリタ

型安全性は

Agda

Coq

による証明

作業量が多く、ミスしやすい

# 本研究のアプローチ：型安全インタプリタ

対象言語

メタ言語

型システム



型付き抽象構文木

意味論



型安全インタプリタ

インタプリタ実装 = 型安全性の証明

# 型付き抽象構文木[Altenkirch and Reus CSL'99]

対象言語の型付け可能な項を表現

```
data Expr : Ty → Set where
  true   : Expr Bool
  false  : Expr Bool
  num    :  $\mathbb{N}$  → Expr Nat
  succ   : Expr Nat → Expr Nat
```

型システムに対応

$$\frac{n : \text{Nat}}{\text{succ } n : \text{Nat}}$$

```
succ zero : Expr Nat
succ true ➡ type error!
```

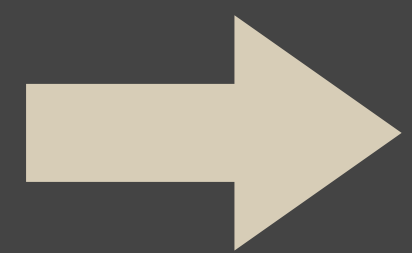
型付け不可能な項は  
型検査で除外

# 型安全インタプリタ[Altenkirch and Reus CSL'99]

対象言語の意味論の記述方法のひとつ

$\text{eval} : \text{Expr } T \rightarrow \text{Val } T$   $\text{Val } T : T$ 型の値を表す型

インタプリタの型 = 型安全性の言明



- インタプリタに対するメタ言語の型検査  
= 対象言語の型安全性の証明

# 実用的な機能による対象言語の拡張

新しい機能のための証明項をインタプリタに追加

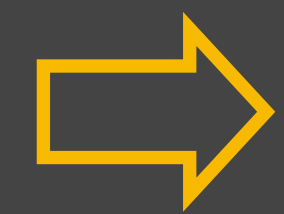
➡ インタプリタが複雑になる

インタプリタの実装を抽象化して証明項を隠蔽

# 例: 可変状態の型安全インタプリタ [Poulsen POPL'18]

$\text{eval} : \text{Expr } \Gamma \ T \rightarrow \text{Env } \Gamma \ \Sigma \rightarrow \text{Store } \Sigma \rightarrow$   
 $\text{Maybe } (\exists \lambda \ \Sigma' \rightarrow \text{Store } \Sigma' \times \text{Val } T \ \Sigma' \times \Sigma \sqsubseteq \Sigma')$

証明項: 可変状態の数が増える



$\text{eval} : \text{Expr } \Gamma \ T \rightarrow \text{M } \Gamma \ (\text{Val } T) \ \Sigma$

抽象化

# 研究目標

エフェクト(副作用)やコエフェクトを持つ言語の  
型安全インタプリタの簡潔な実装を  
支援するライブラリの開発

本論文：エフェクトを持つ言語の型安全インタプリタを実装

⇒ エフェクトのインタプリタに有効な抽象化を分析



# 貢献

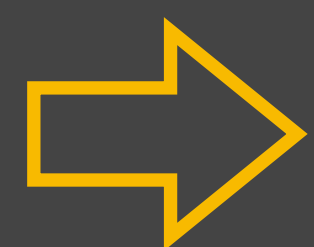
継続を操作するエフェクトを持つ言語の型安全インタプリタを実装

- ・  $\lambda_{s/r}$  : 限定継続命令 [Danvi&Filinski 1990] を持つ言語
- ・  $\lambda_e$  : 代数的エフェクトとハンドラ [Pretnar 2015] を持つ言語

問題：第一級継続の取り扱い

パラメトリック高階抽象構文 (PHOAS) [Chlipala ICFP'08]

+ 継続渡しスタイル (CPS) [Plotkin 1975]



共にCPS意味論の書き下しとしてインタプリタ実装可能

# 型付き抽象構文木の実装：PHOAS

対象言語の変数束縛をメタ言語の束縛機構で表現

構文定義

```
data Expr (Var : VTy → Set) : VTy → Set where
  var : Var T → Expr Var T
  fun : (Var A → Expr Var B) → Expr Var (A ⇒ B)
```

項の例( $\lambda x.x$ )

```
id : Expr Var (A ⇒ A)
id = fun (λ x → var x)
```

# 継続渡しスタイルとCPS意味論

継続渡しスタイルのインタプリタは継続を明示的に受け取る

(例)数値の評価

`eval (num n) k = k n`

継続：その時点での残りの計算

CPS意味論：項と継続を受け取る評価関数  $\xi$  による意味論定義

(例)数値の評価

$$\xi[n]\kappa = \kappa(n)$$

# $\lambda_e$ の型付け

`let x = do choose() in return x`

`: A ! {choose : Unit -> A}`

計算結果の値の型

処理すべきオペレーションの集合

`handle (let x = do choose() in return x)  
with { return x -> return x  
      choose _ k -> k true }`

`: Bool ! {}`

# $\lambda_e$ の評価

安全なプログラムはすべてのオペレーションをハンドラで処理する

✓ `handle (let x = do choose() in return x)`  
`with { return x -> return x`  
`choose _ k -> k true }`

(\* 結果 : true \*)

✗ `let x = do choose() in return x`

動作が未定義のオペレーション

# $\lambda_e$ の型付き抽象構文木

PHOASで実装

値の型  $A$

```
data Val (Var : VTy -> Set) : VTy -> Set where
  var      : Var T -> Val Var T
  fun      : (Var A -> Cmp Var C) -> Val Var (A => C)
```

計算の型  $(A!E)$

```
data Cmp (Var : VTy -> Set) : CTy -> Set where
  Return    : Val Var A -> Cmp Var (A , E)
  Let_In_   : Cmp Var (A , E) ->
    (Var A -> Cmp Var (B , E)) -> Cmp Var (B , E)
```

# $\lambda_e$ の型安全インタプリタ

型安全性：  $A$ 型の純粋な計算は $A$ 型の値に評価される

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

$A! \{ \}$  型

トップレベルプログラムの制約：

すべてのオペレーションをハンドラによって処理することを表現

# $\lambda_e$ の型安全インタプリタの構成

- `eval` はトップレベルプログラムのインタプリタ

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

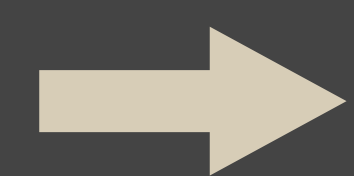
- `evalv`, `evalc`(値、計算に対するインタプリタ)

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



# インタプリタの実装：CPS意味論の書き下し

CPS変換[Hillerstrom FSCD'17]に基づくCPS意味論を構築



インタプリタの実装 = CPS意味論の書き下し


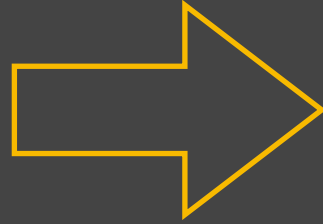
計算の評価関数

値の評価関数

CPS意味論  $\xi_c$   $[do\ l\ V]\rho\kappa h = h(l, \xi_v[V]\rho, \lambda x. \kappa\ x\ h)$

```
evalc (Do l v) k h
= h l (evalv v) (λ x -> k x h)
```

# 今後の課題

- エフェクトを持つ言語の型安全インタプリタのライブラリ化  
 $\lambda_{s/r}$ ,  $\lambda_e$ の型安全インタプリタの共通構造：継続渡しスタイル  
`evalc c k h`  モナドによる隠蔽が可能  
 継続 $k$ ・ハンドラ $h$ を直接扱わず意味論を記述可能
- コエフェクトを持つ言語のための  
型付き抽象構文木・型安全インタプリタを実装
  - Quantitative types<sup>[Atkey LICS'18]</sup>