

エフェクトハンドラを持つ言語に対する 依存型付きコンパイラ

津山勝輝 叢悠悠 増原英彦

東京工業大学

背景 / 型付き構文木^[Altenkirch+'99]

実装対象言語の型付け可能な項のみを表す構文木

BoolNatのNat型の項を表す型

num 2 : Exp Nat bool true : Exp Bool

add (num 1) (num 2) : Exp Nat

BoolNatの型付き構文木

```
data Exp : Ty → Set where
  num    : ℕ → Exp Nat
  bool   : ℬ → Exp Bool
  add    : Exp Nat → Exp Nat → Exp Nat
```

ホスト言語: Agda

背景 / 型付き構文木^[Altenkirch+'99]

実装対象言語の型付け可能な項のみを表す構文木

○ `add (num 1) (num 2)`
✗ `add (bool true) (num 2)`

BoolNatの型付き構文木

```
data Exp : Ty → Set where
  num    : ℕ → Exp Nat
  bool   : ℬ → Exp Bool
  add    : Exp Nat → Exp Nat → Exp Nat
```

ホスト言語: Agda

背景 / 型付き構文木^[Altenkirch+'99]

実装対象言語の型付け可能な項のみを表す構文木

○ `add (num 1) (num 2)`
✗ `add (bool true) (num 2)`

BoolNatの型付き構文木

```
data Exp : Ty → Set where
  num    : ℕ → Exp Nat
  bool   : ℬ → Exp Bool
  add    : Exp Nat → Exp Nat → Exp Nat
```

BoolNatの型付け判断を
ホスト言語の型検査で行う

$$\frac{x : \text{Nat} \quad y : \text{Nat}}{\text{add } x \ y : \text{Nat}} \quad [\text{T-ADD}]$$

ホスト言語: Agda

背景 / 型付き構文木^[Altenkirch+'99]

実装対象言語の型付け可能な項のみを表す構文木

○ `add (num 1) (num 2)`
✗ `add (bool true) (num 2)`

BoolNatの型付き構文木

```
data Exp : Ty → Set where
  num  : ℕ → Exp Nat
  bool  : ℬ → Exp Bool
  add   : Exp Nat → Exp Nat → Exp Nat
```

BoolNatの型付け判断を
ホスト言語の型検査で行う

$$\frac{x : \text{Nat} \quad y : \text{Nat}}{\text{add } x \ y : \text{Nat}} \quad [\text{T-ADD}]$$

ホスト言語: Agda

背景 / 依存型付きコンパイラ [Pickard+'21]

ホスト言語で `compile` に型がつく

\iff `compile` は型を保存する

`compile` : $\text{Exp } T \rightarrow \text{Code } S \text{ (} T :: S \text{)}$

ソース言語の型付き構文木

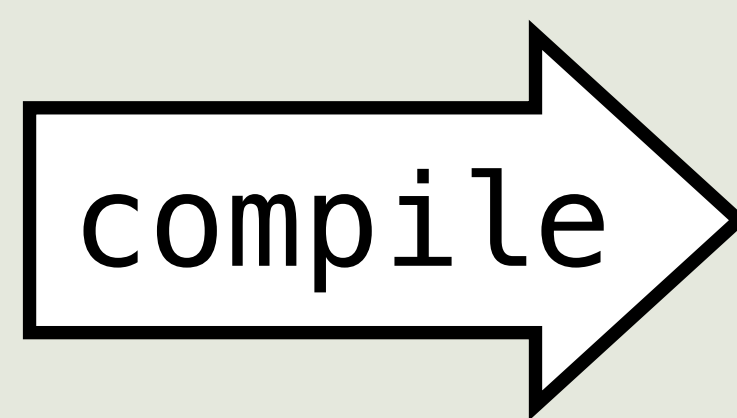
ターゲット言語の型付き構文木

背景 / 依存型付きコンパイラ [Pickard+'21]

ホスト言語で `compile` に型がつく
 \iff `compile` は型を保存する

$$\text{compile} : \text{Exp } T \rightarrow \text{Code } S \quad (T :: S)$$

ソース言語の型付き構文木
`add (num 1) (num 2)`
: `Exp Nat`



ターゲット言語の型付き構文木
`PUSH 1 >> PUSH 2 >> ADD`
: `Code S (Nat :: S)`

研究目標

エフェクトハンドラを持つ言語の
依存型付きコンパイラを実装する

Q

第一級継続を扱う言語の依存型付きコンパイラを
実装するときに何が必要なのか？

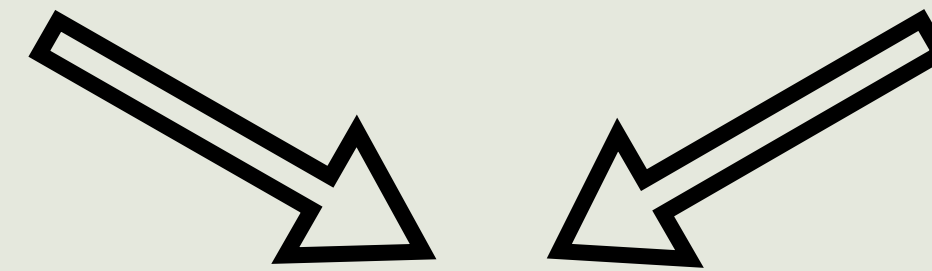
貢献

[Bahr+'22]

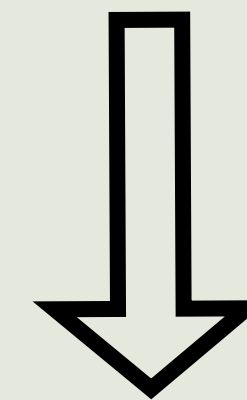
ラムダ計算のコンパイラ

[Pickard+'21]

例外処理のコンパイラ



1. λ_{ex} (ラムダ計算 + 例外処理機構)のコンパイラ



A

第一級継続を扱うため
スタック多相の導入

貢献 — 2. λ_{eff} (エフェクトハンドラを持つ言語)のコンパイラ

λ_{ex} : ラムダ計算 + 例外処理機構

data Cmp : CTy \rightarrow Set where

...

throw : ...

catch : ...

λ_{ex}

data Code : StackTy \rightarrow StackTy \rightarrow Set where

...

THROW : ...

MARK : ...

ターゲット言語

λ_{ex} : ラムダ計算 + 例外処理機構

data Cmp : CTy \rightarrow Set where

...

throw : ...

catch : ...

λ_{ex}

data Code : StackTy \rightarrow StackTy \rightarrow Set where

...

THROW : ...

MARK : ...

ターゲット言語

コンパイル例

catch

(0 + **throw**)

1

-- 結果 1

compile

例外ハンドラのもとで実行

MARK

(**PUSH 1 HALT**)

(**PUSH 0 THROW**)

例外を発生させる

スタックベース抽象機械の例外処理機構[Pickard+'21]

実行される命令

スタック

S

MARK (PUSH 1 HALT) (PUSH 0 THROW)

スタックベース抽象機械の例外処理機構[Pickard+'21]

実行される命令

MARK (PUSH 1 HALT) (PUSH 0 THROW)

(PUSH 0 THROW)

例外ハンドラ **(PUSH 1 HALT)**
をスタックにプッシュ

スタック

S

PUSH 1 HALT S

スタックベース抽象機械の例外処理機構[Pickard+'21]

実行される命令

MARK (PUSH 1 HALT) (PUSH 0 THROW)

(**PUSH 0** THROW)

THROW

スタック

S

PUSH 1 HALT S

0 PUSH 1 HALT S

スタックベース抽象機械の例外処理機構[Pickard+'21]

実行される命令

MARK (PUSH 1 HALT) (PUSH 0 THROW)

(PUSH 0 THROW)

0 を破棄し
例外ハンドラを実行

THROW

PUSH 1 HALT

スタック

S

PUSH 1 HALT S

0 PUSH 1 HALT S

S

スタックベース抽象機械の例外処理機構[Pickard+'21]

実行される命令

MARK (PUSH 1 HALT) (PUSH 0 THROW)

(PUSH 0 THROW)

THROW

PUSH 1 HALT

スタック

S

PUSH 1 HALT S

0 PUSH 1 HALT S

S

1 S

スタックベース抽象機械の例外処理機構[Pickard+'21]

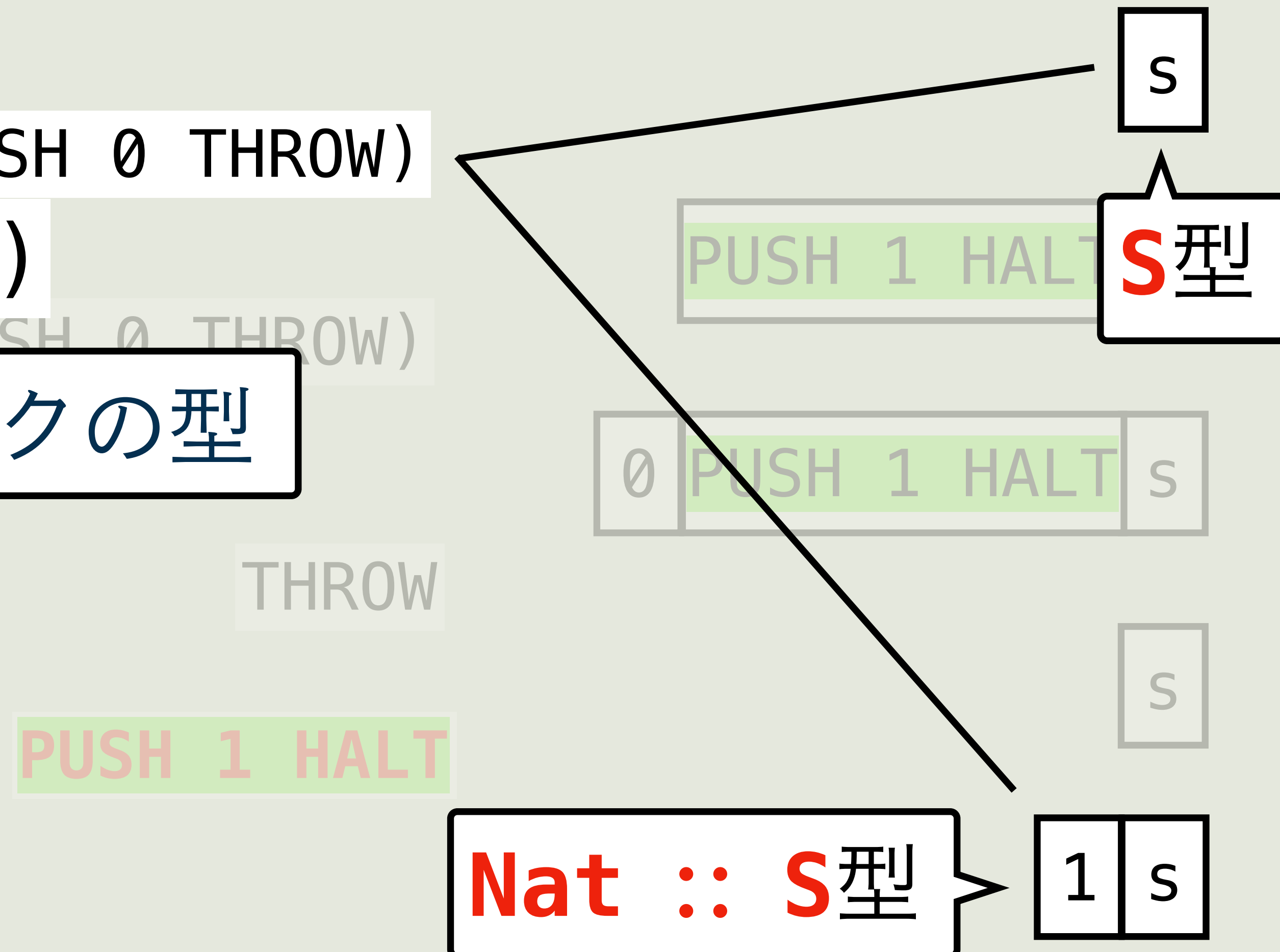
実行される命令

MARK (PUSH 1 HALT) (PUSH 0 THROW)

: Code **S** (**Nat :: S**)

実行前後のスタックの型

スタック



スタックベース抽象機械の例外処理機構 [Pickard+'21]

実行される命令

スタック

MARK (PUSH 1 HALT) (PUSH 0 THROW)

Code S (Nat :: S)

(PUSH 0 THROW)

PUSH 1 HALT S

THROW

PUSH 1 HALT

0 PUSH 1 HALT S

S

1 S

スタックベース抽象機械の例外処理機構 [Pickard+'21]

実行される命令

MARK (PUSH 1 HALT) (PUSH 0 THROW)

(PUSH 0 THROW)

THROW

PUSH 1 HALT

スタック

Code **S (Nat :: S)**

PUSH 1 HALT S

HandTy S (Nat :: S) :: S
型のスタック

S

1 S

スタックベース抽象機械の例外処理機構 [Pickard+'21]

実行される命令

MARK (PUSH 1 HALT) (PUSH 0 THROW)

(PUSH 0 THROW)

Code
 (**HandTy S (Nat :: S) :: S**)
 (Nat :: S)

スタック

Code S (Nat :: S)

PUSH 1 HALT S

HandTy S (Nat :: S) :: S
型のスタック

S

1 S

例外処理機構のみのコンパイラ [Pickard+'21]

例外を起こす可能性のある項に対するコンパイラ

compile? :

$\frac{\text{Exp } (A, a) \rightarrow$

$\text{Code } (\text{ValTy } A :: (S_1 ++ \text{HandTy } S \ S' :: S)) \ S' \rightarrow$

$\text{Code } (S_1 ++ \text{HandTy } S \ S' :: S) \ S'$

ソース言語の項

ターゲット言語の項

例外処理機構のみのコンパイラ [Pickard+'21]

例外を起こす可能性のある項に対するコンパイラ

compile? :

Exp (A , a) →

Code (ValTy A :: (S₁ ++ HandTy S S' :: S)) S' →

Code (S₁ ++ HandTy S S' :: S) S'

コード継続

継続は破棄される

compile? throw **k** = THROW

compile? (add e1 e2) **k** = compile? e1 \$ compile? e2 \$ ADD k

add e1 e2 の継続

e1の継続

例外処理機構のみのコンパイラ [Pickard+'21]

例外を起こす可能性のある項に対するコンパイラ

```
compile? :  
  Exp (A , a) →  
  Code (ValTy A :: (S1 ++ HandTy S S' :: S)) S' →  
  Code (S1 ++ HandTy S S' :: S) S'
```

ソース項が例外を起こす場合があるので
実行前スタックにハンドラの存在を要求

例外処理機構のみのコンパイラ [Pickard+'21]

例外を起こす可能性のある項に対するコンパイラ

```
compile? :  
  Exp (A , a) →  
  Code (ValTy A :: (S1 ++ HandTy S S' :: S)) S' →  
  Code (S1 ++ HandTy S S' :: S) S'
```

ソース項と同じ型の値がプッシュされたスタック

λ_{ex} (ラムダ計算 + 例外処理機構)のコンパイラ

ソース・ターゲット言語の型付けに型環境の情報を追加

```
data Val ( $\Gamma$  : Ctx) : VTy  $\rightarrow$  Set
```

```
val :  $\mathbb{N} \rightarrow \text{Val } \Gamma \text{ Nat}$ 
```

```
var :  $A \in \Gamma \rightarrow \text{Val } \Gamma A$ 
```

```
lam : ...
```

```
data Cmp ( $\Gamma$  : Ctx) : CTy  $\rightarrow$  Set
```

```
...
```

```
app : Val  $\Gamma$  Nat  $\rightarrow$  Val  $\Gamma$  Nat  $\rightarrow$  Cmp  $\Gamma$  (Nat , false)
```

値を表す型付き構文木

計算を表す型付き構文木

λ_{ex} (ラムダ計算 + 例外処理機構) のコンパイラ

ソース・ターゲット言語の型付けに型環境の情報を追加

```
data Val ( $\Gamma$  : Ctx) : VTy  $\rightarrow$  Set
```

```
val :  $\mathbb{N} \rightarrow \text{Val } \Gamma \text{ Nat}$ 
```

```
var :  $A \in \Gamma \rightarrow \text{Val } \Gamma A$ 
```

```
lam : ...
```

```
data Cmp ( $\Gamma$  : Ctx) : CTy  $\rightarrow$  Set
```

```
...
```

```
app : Val  $\Gamma$  Nat  $\rightarrow$  Val  $\Gamma$  Nat  $\rightarrow$  Cmp  $\Gamma$  (Nat , false)
```

値を表す型付き構文木

計算を表す型付き構文木

```
compileC? :
```

```
  Cmp  $\Gamma$  (A , a)  $\rightarrow$ 
```

```
  Code  $\Gamma$  (ValTy A :: (S1 ++ HandTy  $\Gamma_1$  S S' :: S)) S'  $\rightarrow$ 
```

```
  Code  $\Gamma$  (S1 ++ HandTy  $\Gamma_1$  S S' :: S) S'
```

λ_{eff} : エフェクトハンドラを持つソース言語

λ_{eff} オペレーション呼び出しをハンドラが処理する

```
handle (let x = do op () in 2 + x)
with handler {...; op _ k -> 1 + (k 0) }
→* 1 + (2 + 0) → 3
```

ハンドラは継続を使用可能

コード継続の型をスタック多相にすることで
第一級継続を扱うことができる

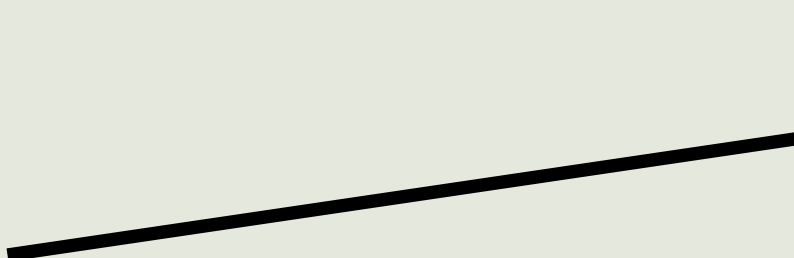
ターゲット言語における第一級継続の捕捉・再開

実行命令に対応するソース項

スタック

```
handle (let x = do op () in 2 + x)
with handler {...; op _ k -> 1 + (k 0) }
```

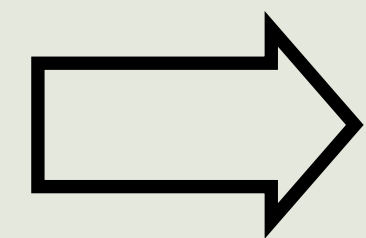
S



ターゲット言語における第一級継続の捕捉・再開

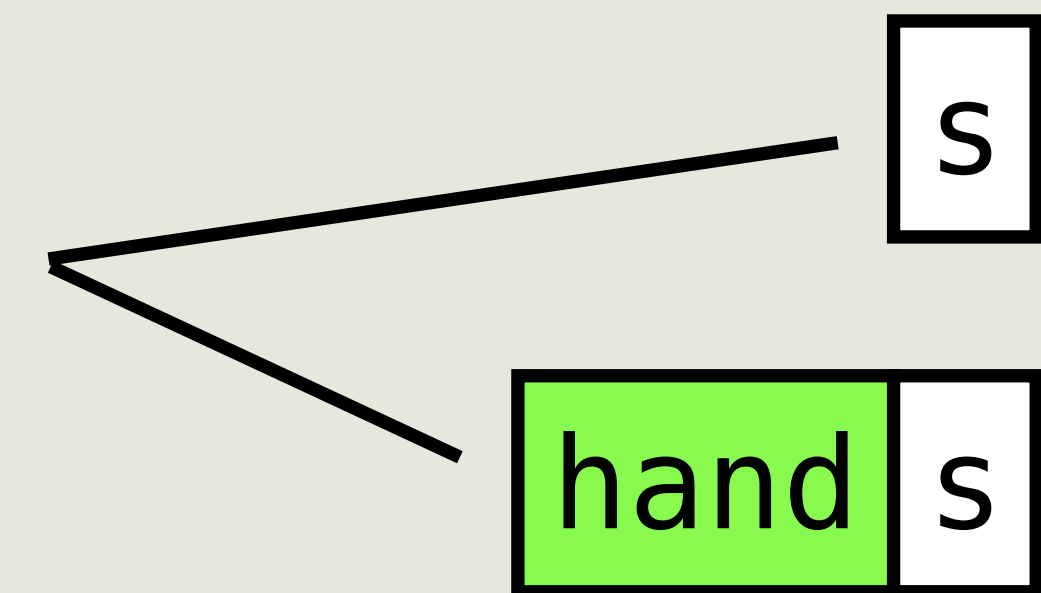
実行命令に対応するソース項

```
handle (let x = do op () in 2 + x)  
with handler {...; op _ k -> 1 + (k 0) }
```



```
let x = do op () in 2 + x
```

スタック

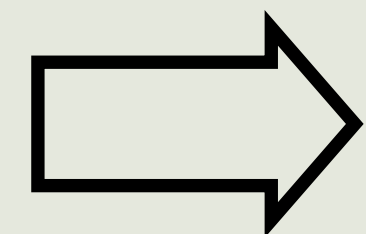


ターゲット言語における第一級継続の捕捉・再開

実行命令に対応するソース項

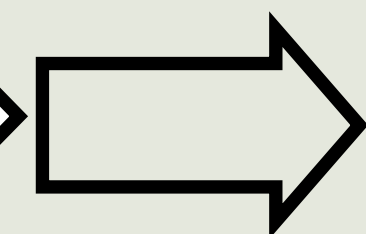
スタック

```
handle (let x = do op () in 2 + x)
with handler {...; op _ k -> 1 + (k 0)}
```



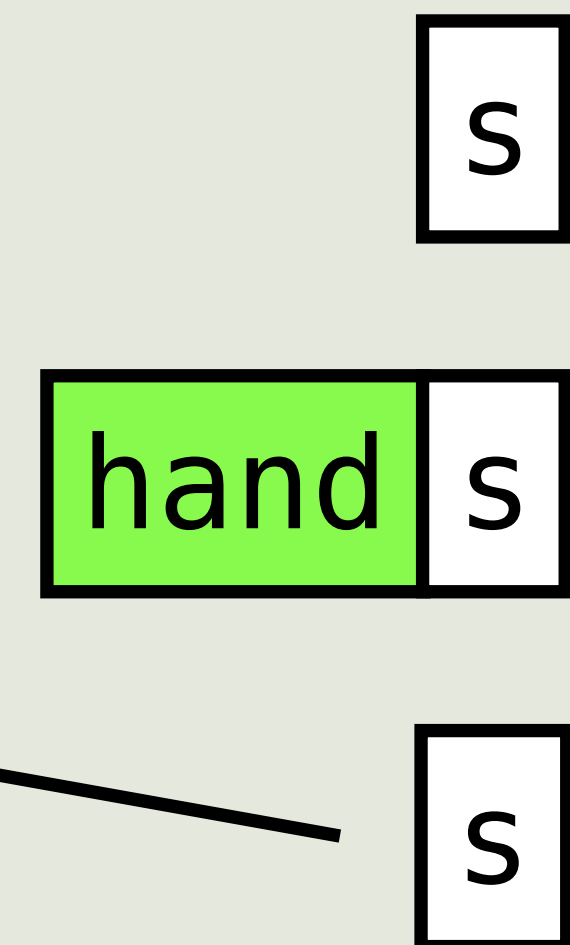
```
let x = do op () in 2 + x
```

継続を捕捉



1 + (k 0)

```
handle (let x = □ in 2 + x)
with hand
```



ターゲット言語における第一級継続の捕捉・再開

実行命令に対応するソース項

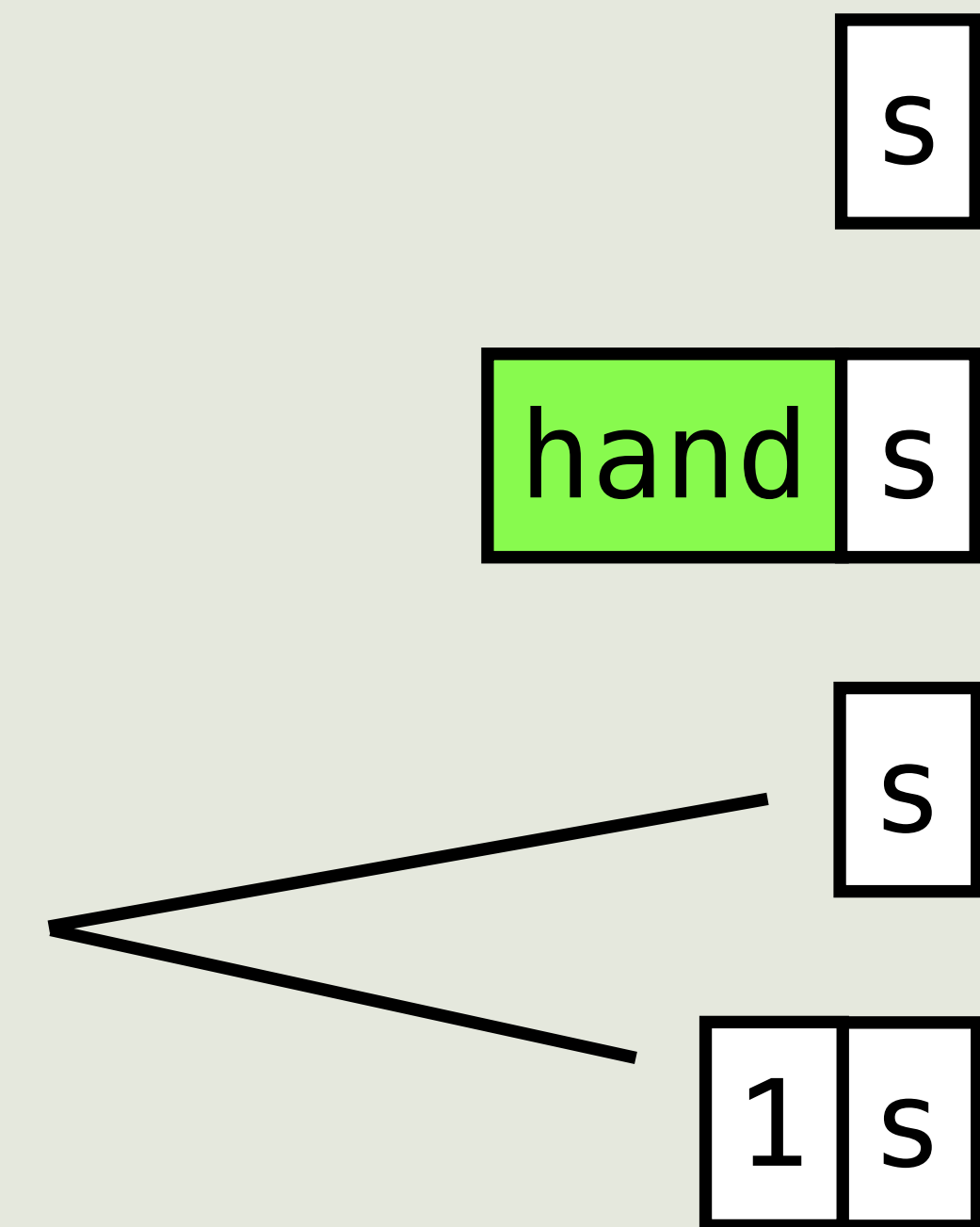
```
handle (let x = do op () in 2 + x)
with handler {...; op _ k -> 1 + (k 0) }
```

⇒ let x = do op () in 2 + x

⇒ 1 + (k 0)

⇒ k 0

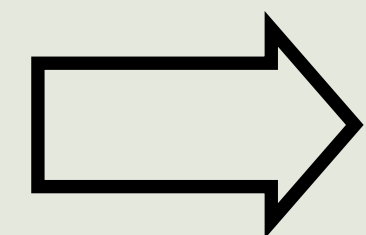
スタック



ターゲット言語における第一級継続の捕捉・再開

実行命令に対応するソース項

```
handle (let x = do op () in 2 + x)
with handler {...; op _ k -> 1 + (k 0) }
```



```
let x = do op () in 2 + x
```

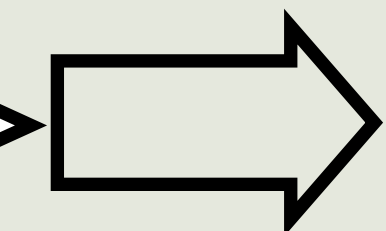


```
handle (let x = □ in 2 + x)
with hand
```

```
1 + (k 0)
```

k 0

継続を再開



```
let x = 0 in 2 + x
```

スタック

s

hand s

s

1 s

hand 1 s

スタック多相の必要性

実行命令に対応するソース項

スタック

```
handle (let x = do op () in 2 + x)
with handler {...; op _ k -> 1 + (k 0) }
```

コード継続を捕捉

```
let x = do op () in 2 + x
```

hand

s



```
λx.handle (2 + x) with handler {...; op _ k -> 1 + (k 0) }
```

```
1 + (k 0)
```

コード継続を再開

hand

1

s

1

s

s

s

継続を再開



スタック多相の必要性

実行命令に対応するソース項

スタック

HandTy Γ **S** (**N** :: **S**)

hand s

let x = do op () in 2 + x

1 + (k 0)

hand 1 s

継続を再開

let x = 0 in 2 + x

スタック多相の必要性

実行命令に対応するソース項

スタック

HandTy Γ **S** $(N :: S)$ 型の
ハンドラで処理されるコード継続

HandTy Γ **S** $(N :: S)$

let x = do op () in 2 + x

hand s

1 + (k 0)

$\lambda x. \text{handle } (2 + x) \text{ with } h$

継続を再開

let x = 0 in 2 + x

hand 1 s

スタック多相の必要性

ハンドラの実行前スタックの型が変化する可能性がある

HandTy Γ **S** $(N :: S)$ 型の
ハンドラで処理されるコード継続

HandTy Γ **S** $(N :: S)$

let x = do op () in 2 + x

hand s

1 + (k 0)

HandTy Γ $(N :: S)$ $(N :: S)$

hand 1 s

スタック多相の必要性

ハンドラの実行前スタックの型が変化する場合がある

~~HandTy Γ S ($N :: S$) 型の
ハンドラで処理されるコード継続~~

let x = do op () in 2 + x

HandTy Γ S ($N :: S$) 型のハンドラにも
HandTy Γ ($N :: S$) ($N :: S$) 型のハンドラにも
処理されうるコード継続

hand s

1 s

スタック多相の導入

コード継続の型

λ_{ex} Code Γ (ValTy $A :: S_1 ++$ (HandTy Γ_1 **S** **S'**) $:: S$) **S'**

λ_{eff} **$\forall \{S S'\}$** \rightarrow
Code Γ (ValTy $A :: S_1 ++$ (HandTy Γ_1 **S** **S'** C) $:: S$) **S'**

HandTy Γ **S** $(N :: S)$ 型のハンドラにも
HandTy Γ **$(N :: S)$** $(N :: S)$ 型のハンドラにも
処理されるコード継続

λ_{eff} 依存型付きコンパイラ

ContCode $\Gamma \ S_1 \ (B, E) = \forall\{S \ S'\} \rightarrow$
Code $\Gamma \ (\text{ValTy } A :: S_1 \ ++ \ (\text{HandTy } \Gamma_1 \ S \ S' \ C) :: S) \ S'$

計算に対するコンパイラの型定義

compileC : Cmp $\Gamma \ (A, E) \rightarrow$
ContCode $\Gamma \ (\text{ValTy } A :: S_1) \ (A', E) \rightarrow$
Code $\Gamma \ (S_1 \ ++ \ \text{HandTy } \Gamma_1 \ S \ S' \ (A', E) :: S) \ S'$

compileC (Do op v) **k** = compileV v (**CALLOP** op **k**)

継続を捕捉しオペレーションを呼び出す命令

まとめ

λ_{eff} に対する依存型付きコンパイラの実装

Q

第一級継続を扱う言語の依存型付きコンパイラを実装するときに何が必要なのか？

A

コード継続の型を変更しスタック多相にする

今後の課題

- コンパイラ計算[Pickard+'21]により動的意味の保存を保証
 - $\text{exec}(\text{compile } e) \ s = (\text{eval } e) :: s$
- ハンドラ内の継続の使われ方に応じた最適化のサポート
 - 呼び出し回数[multicore ocaml]
 - 末尾位置での呼び出し[Xie+'21]

まとめ

λ_{eff} に対する依存型付きコンパイラの実装

Q

第一級継続を扱う言語の依存型付きコンパイラを実装するときに何が必要なのか？

A

コード継続の型を変更しスタック多相にする