# Current situation of Project-Intrinsically

Syouki Tsuyama (Tokyo Tech)

# Contents

- Research concept & Background
  - One-shot Continuation
  - Multiplicity
  - Dependently-typed Compilers
- Current Condition & Future Work

# Contents

- Research concept & Background
  - One-shot Continuation
  - Multiplicity
  - Dependently-typed Compilers
- Current Condition & Future Work

# Research Concept

Effect system + Tracing variable usage

⇩

Detect how many times operations use continuations

⇩

Make compiler that optimize based on continuation usage (or optimizer)

# One-shot Continuation[Bruggeman PLDI'96]

Restricted continuations which can be called at most once

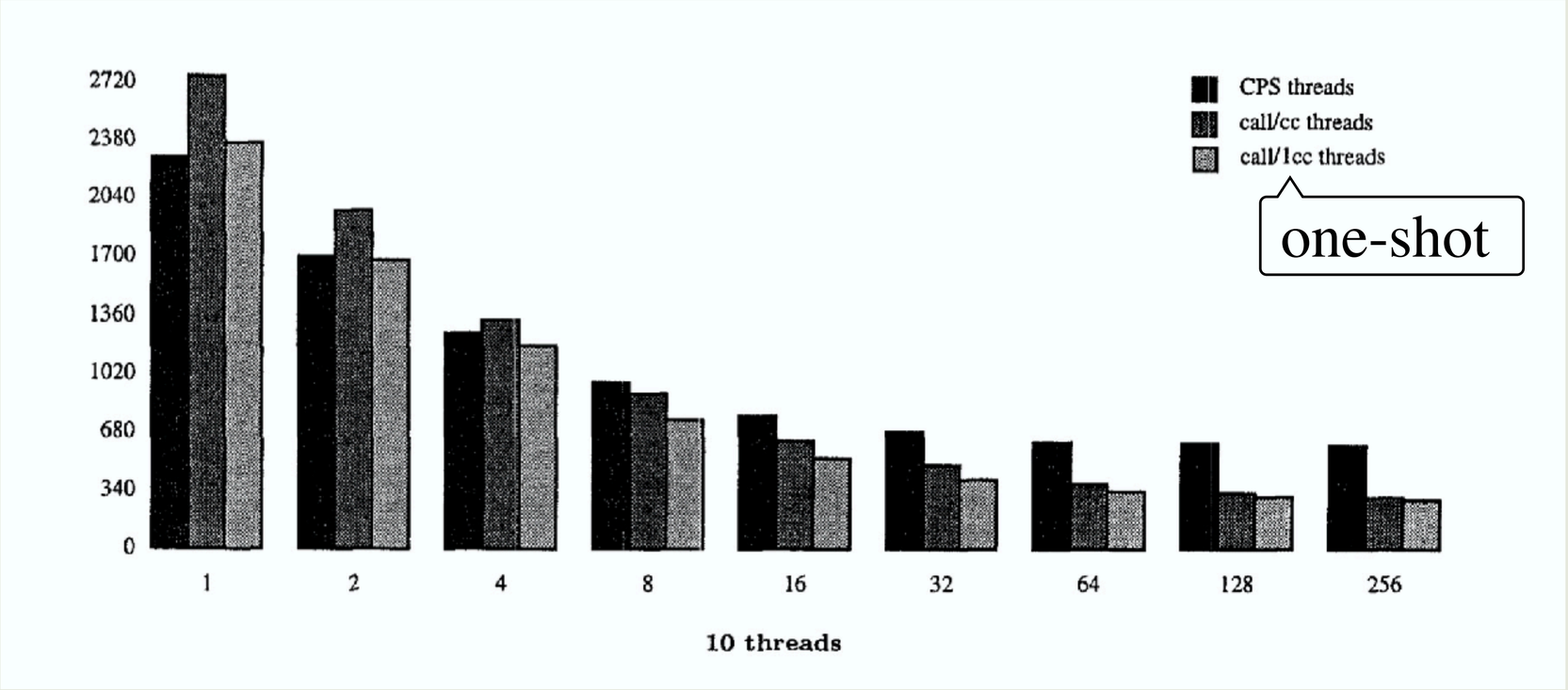
✅ `shift (fun k -> 0)`


✅ `shift (fun k -> k 0)`

❌ `shift (fun k -> (k , k))`

# Compilation for One-shot Continuation

More efficient compilation[Bruggeman PLDI'96] than unrestricted continuation



# Multiplicity

Multiplicity is given to each variable

to indicate how many times the variable is used

e.g.

| | |
|---|---|
| `0` | Unused |
| `1` | Used at once |
| `ω` | Unrestricted |

# Multiplicity Example

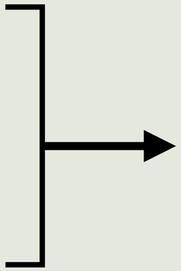
```
drop : (0 x : Int) -> Int
drop _ = 0

id   : (1 x : Int) -> Int
id x = x

copy : (x : Int) -> Int
copy x = (x , x)
```

# Counting Usage of Continuation

Continuation + Multiplicity

can detect whether k is one-shot or not

```
shift (fun (0 k) -> 0)
shift (fun (1 k) -> k 0)
```



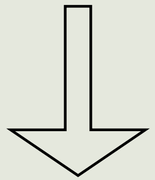
One-shot

```
shift (fun (ω k) -> (k , k))
```

Unrestricted

# Dependently-typed Compiler[Pickard ICFP'21]

Intrinsically-typed AST (IT-AST)



`compile`

Target language

Dependent types help to
derive `compile` which satisfy correctness theorem

# Components of Dependently-typed Compiler

Five components.

1. IT-AST
which represents only well-typed expression of source language

```
data Exp : Ty → Set
  num  : ℕ → Expr ℕ
  bool : 𝔹 → Expr 𝔹
```

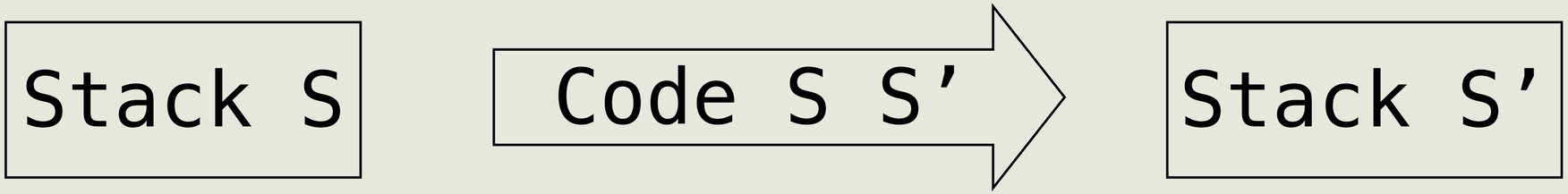
2. ITI
evaluates expressions of type T into values of type T

```
eval : Exp T → T
```

# Components of Dependently-typed Compiler

3. Syntax of target language

Here , `Code S S'` represents stack machine code




```
data Stack : List Ty → Set
data Code : List Ty → List Ty → Set
```

Input stack

Output stack

# Components of Dependently-typed Compiler

4. Excution function

which determines semantics of target language

```
exec : Code S S' → Stack S → Stack S'
```

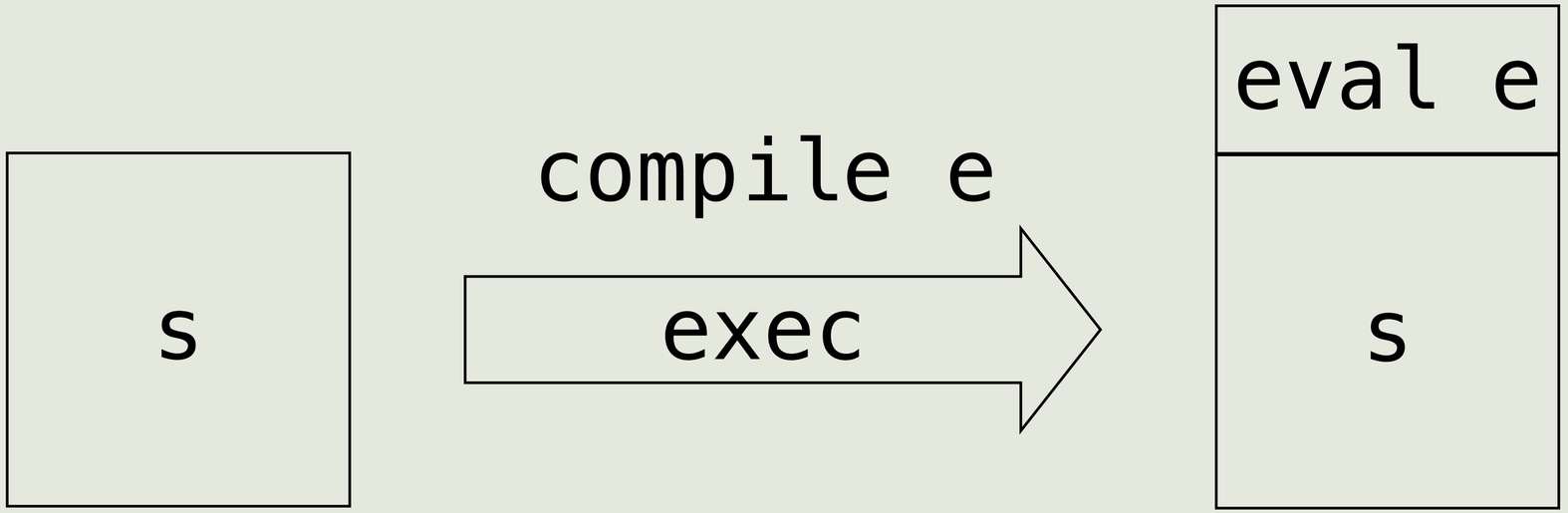
5. Compiler

which transforms IT-AST into target language

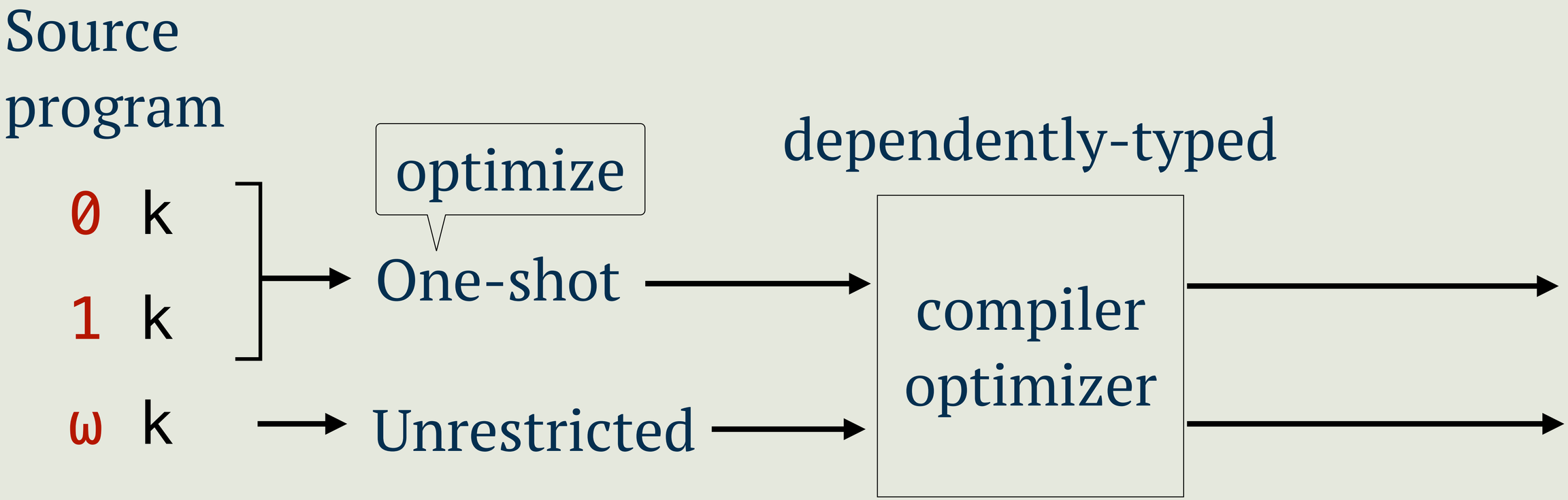
```
compile : Exp T → Code S (T :: S)
```

# Correctness Theorem

```
exec (compile e) s ≡ (eval e) :: s
```



# Summerly


Source
program
0 k
1 k
ω k
optimize
One-shot
Unrestricted
dependently-typed
compiler
optimizer

# Contents

- Research concept & Background
  - One-shot Continuation
  - Multiplicity
  - Dependently-typed Compilers
- Current Condition & Future Work

# λe : A language with algebraic effect

- Effect signature / Definition of effect

```
effect Choose { choose : () → Bool }
```

Operation

- Handler / Behavior of the effect operation

```
handle { if (do choose()) then 1 else 0 }
with { choose _ k → k true }
```

# Adding New Feature / λe + Multiplicity

```
effect Choose { choose : () → Bool ; 1 }
```

Continuation usage

```
handle { if (do choose()) then 1 else 0 }
with { choose _ k → k true }
```

k can be used 1 times

# λe so far

I used PHOAS for variable binding

However, PHOAS cannot extend with multiplicity

```
data Val (Var : VTy → Set) : VTy → Set where
  var : Var T → Val Var T
  fun : (Var A → Cmp Var C) → Val Var (A ⇒ C)
```

e.g. `id = fun (\x → x)`

# Changing PHOAS to De Brujin indice

Rewrited IT-AST using De Brujin indice

```
data Val (Γ : Ctx) : VTy -> Set where
  var  : T ∈ Γ → Val Γ T
  fun  : Cmp (A :: Γ) C → Val Γ (A ⇒ C)
```

e.g. `id = fun (var 0)`

To give multiplicity, I'm going to rewrite to

```
Ctx = List (Ty × Mul)
```

# Future Work

- Complete Intrinsically-typed AST
  - Need to consider composing and separation of `Ctx`
- learn how to compile continuation manipulation[masuko PPL'10]
- Goal : Make a compiler or an optimizer