An Intrinsically Typed Compiler for Algebraic Effect Handlers

Syouki Tsuyama, Youyou Cong, Hidehiko Masuhara
Tokyo Institute of Technology

Goal

Implement type-preserving compilers for languages with first-class continuations

Previous work[Morrisett+'99, Chlipala'07, Pickard+'21] considers

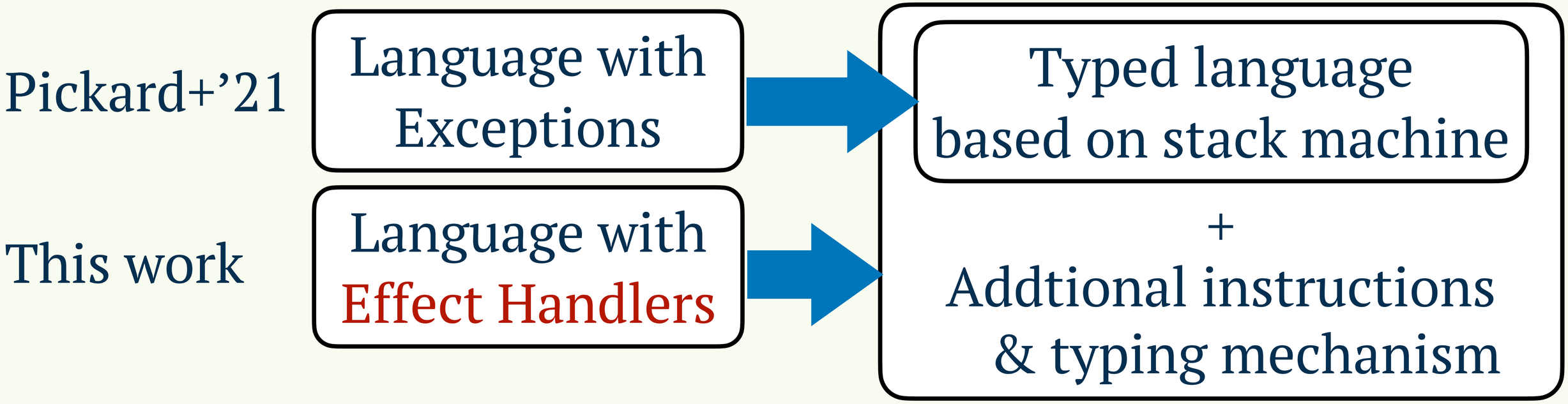
- Functions
- Conditionals & loops
- Exceptions

This work considers

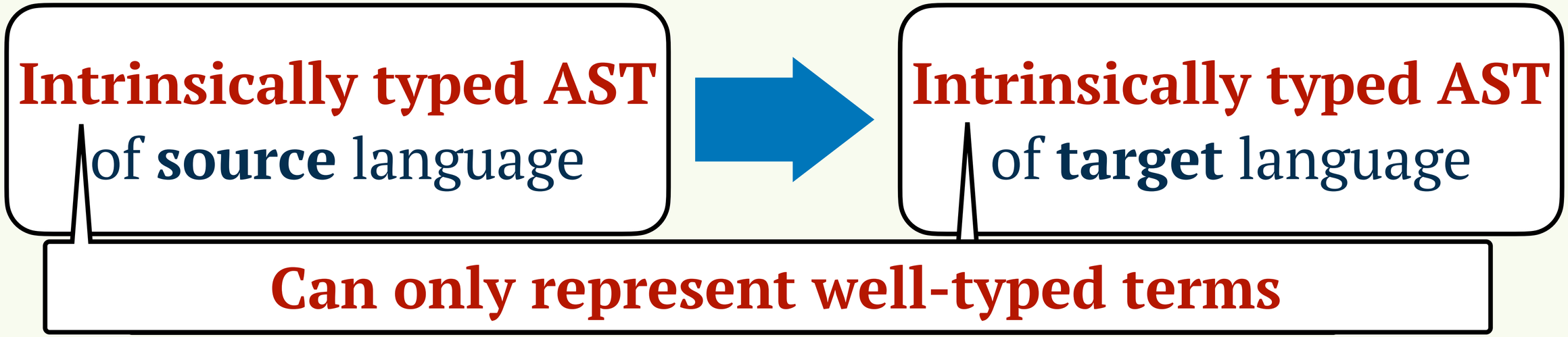
- Effect handlers

Approach

Implement an intrinsically typed compiler using Agda



Background : Intrinsically Typed Compiler



Type checked definition = Proof of type preservation

Background : Intrinsically Typed AST

Simple AST

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

Agda

```
add (num 1) (bool true) : Exp
```

Background : Intrinsically Typed AST

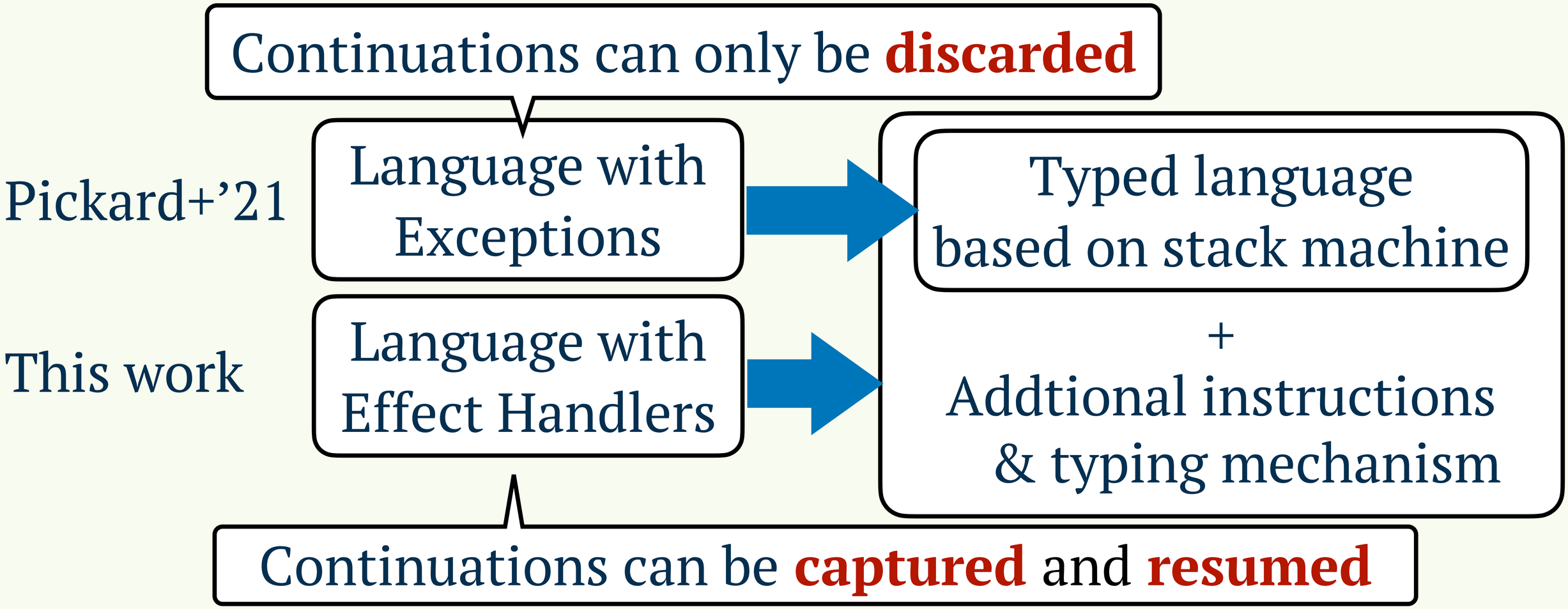
Intrinsically-typed AST

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

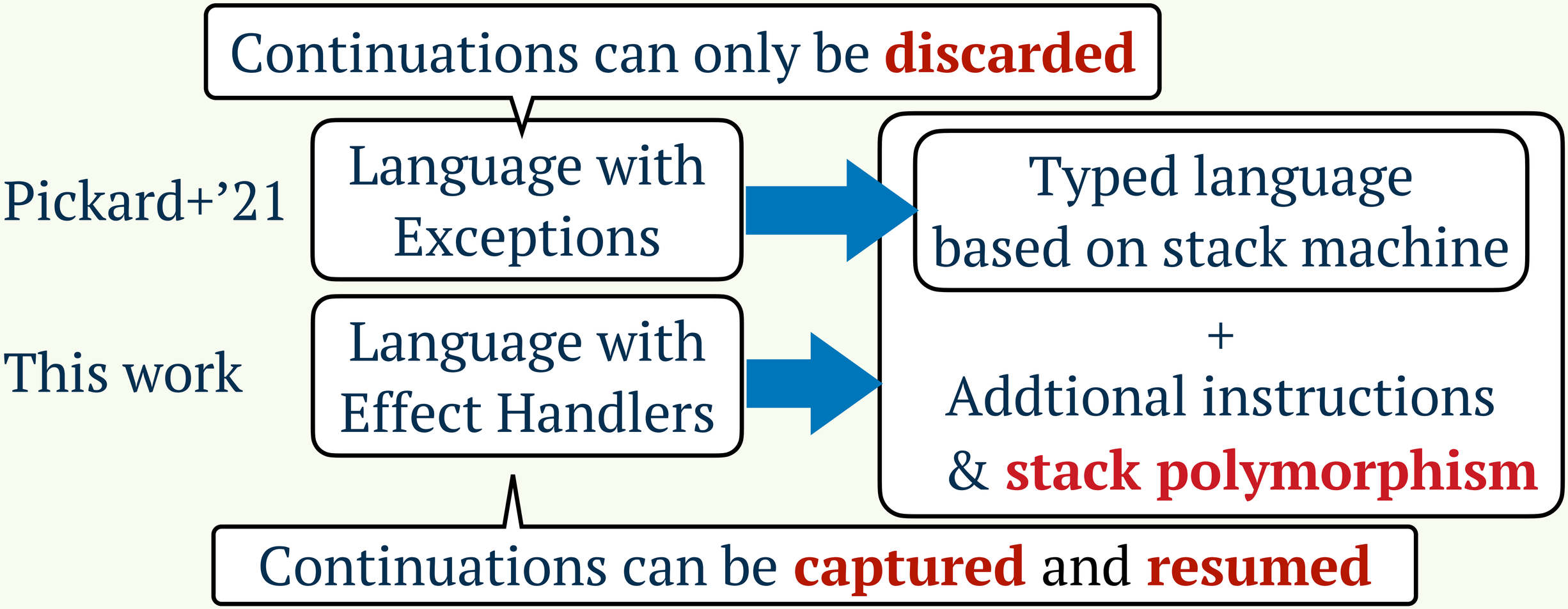
Agda

```
add (num 1) (bool true) : Exp Nat
```

Challenge : Typing Continuations



Solution : Incorporate Stack Polymorphism[Morrisett+'02]



Effect Handlers in Source Language

```
handle 1 + do op ()
with {
  ...
  op _ k -> k 0
}
```

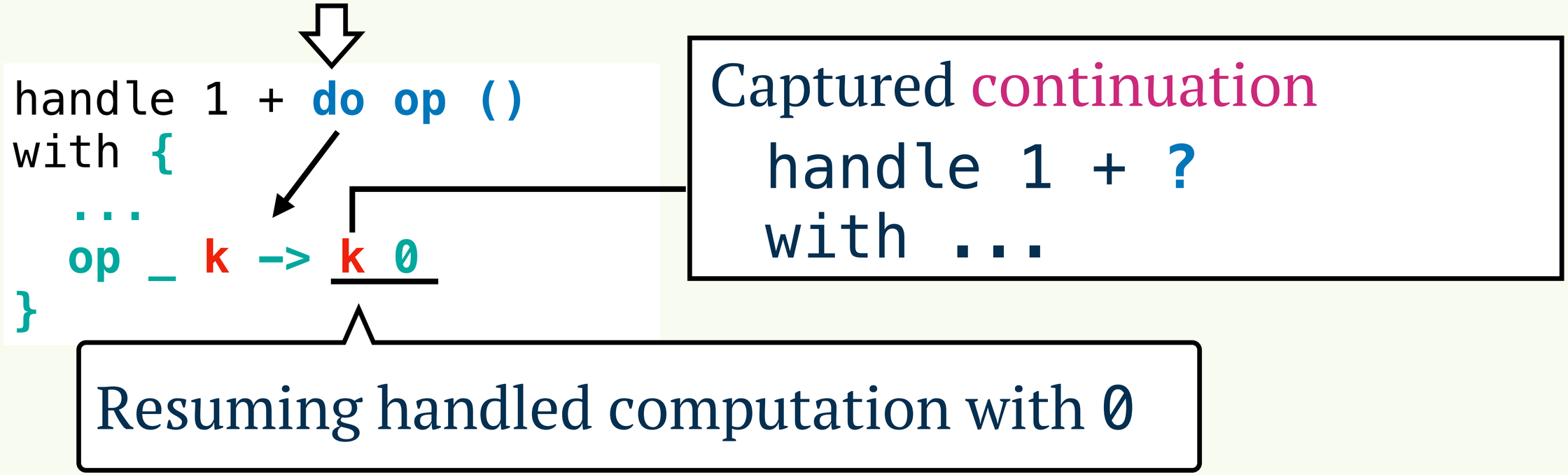
Operations cause effects

Handlers determine behavior of operations

Handlers can use continuaions

Effect Handlers in Source Language

Execution example



Effect Handlers in Source Language

Execution example

```
handle 1 + 0
with {
  x -> x
  op _ k -> k 0
}
```

=> 1

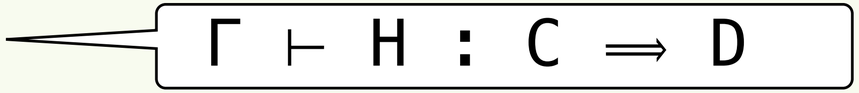
Intrinsically Typed AST of Source Language

Data types representing only well-typed terms

```
data Val (Γ : Ctx) : VTy → Set
```


```
data Cmp (Γ : Ctx) : CTy → Set
```


```
data Hdl (Γ : Ctx) : HTy → Set
```

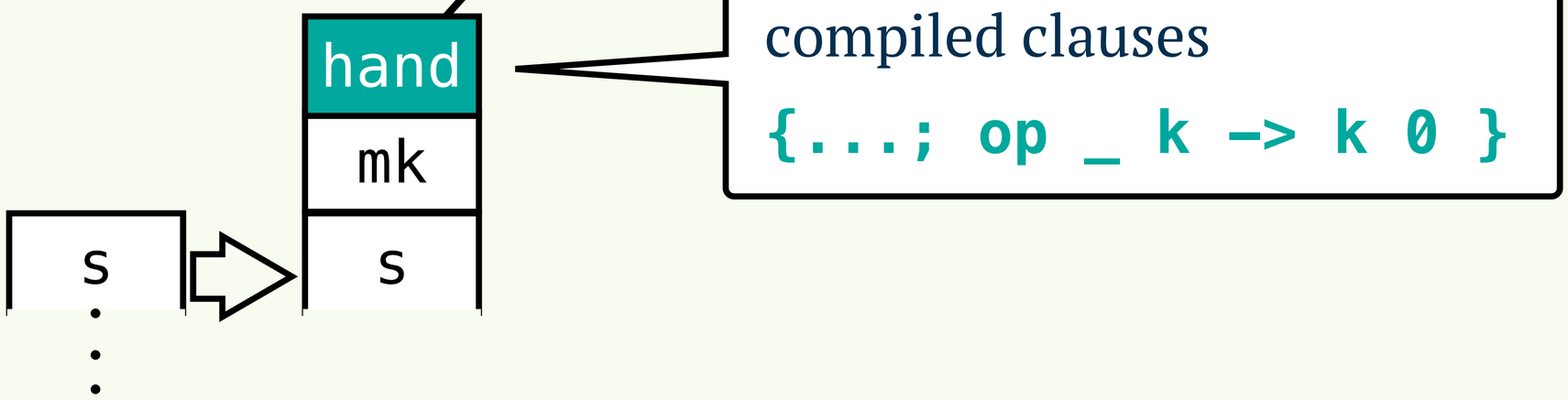


Effect Handlers in Target Language

Source term

```
handle 1 + do op ()
with {...; op _ k -> k 0 }
```

Target stack

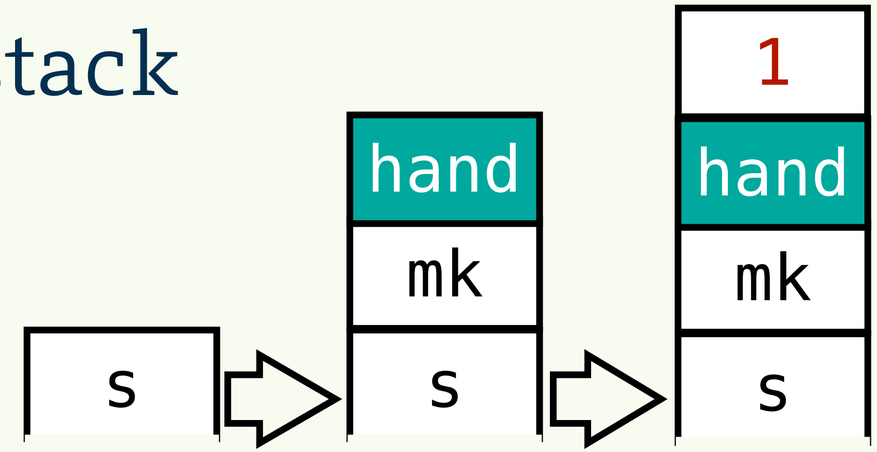


Effect Handlers in Target Language

Source term

```
handle 1 + do op ()
with {...; op _ k -> k 0 }
```

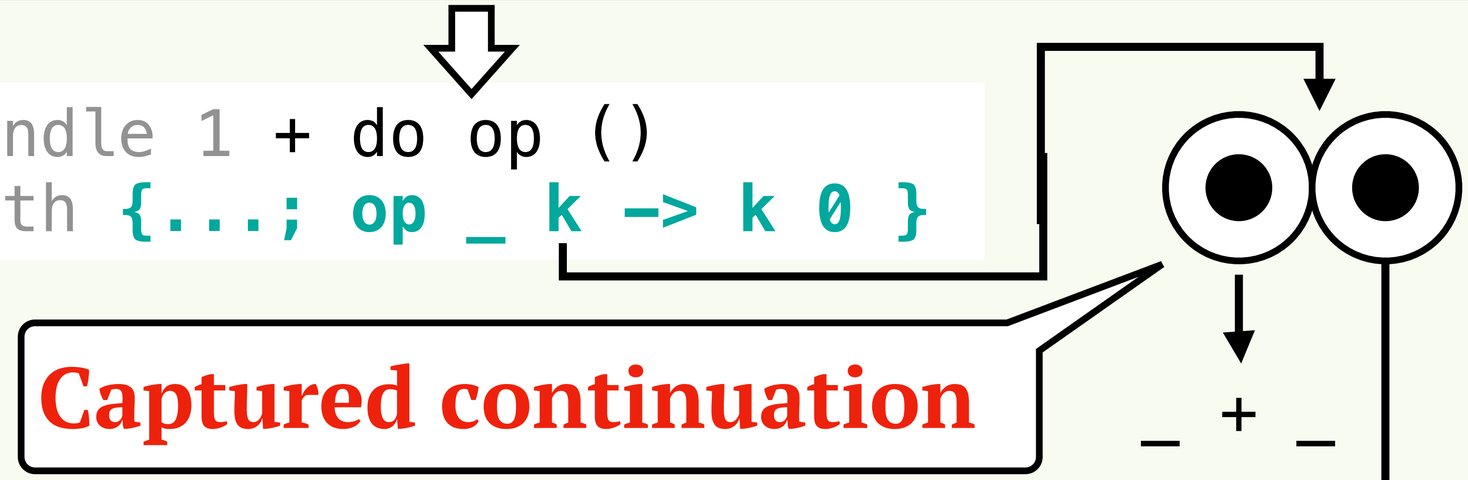
Target stack



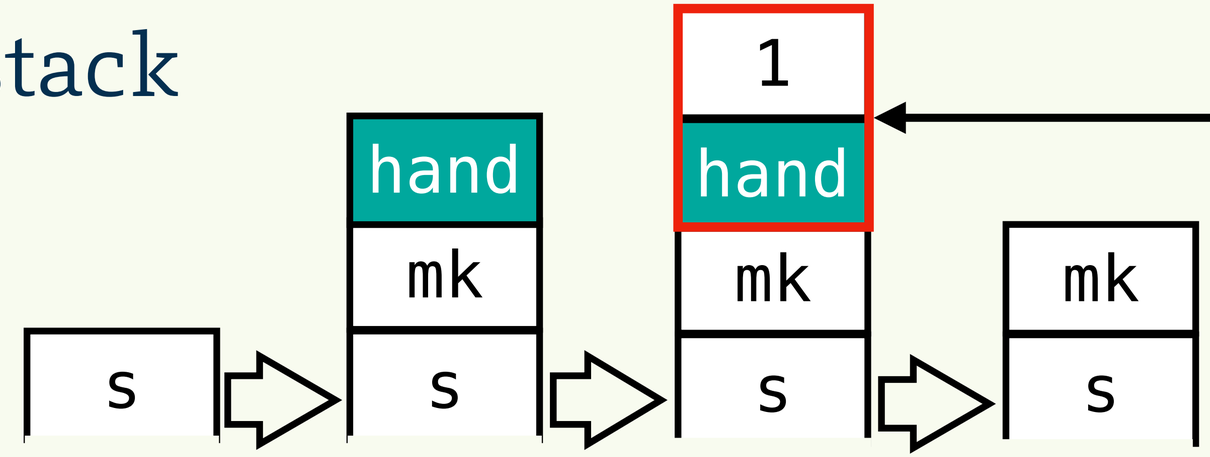
Effect Handlers in Target Language

Source term

```
handle 1 + do op ()
with {...; op _ k -> k 0 }
```



Target stack

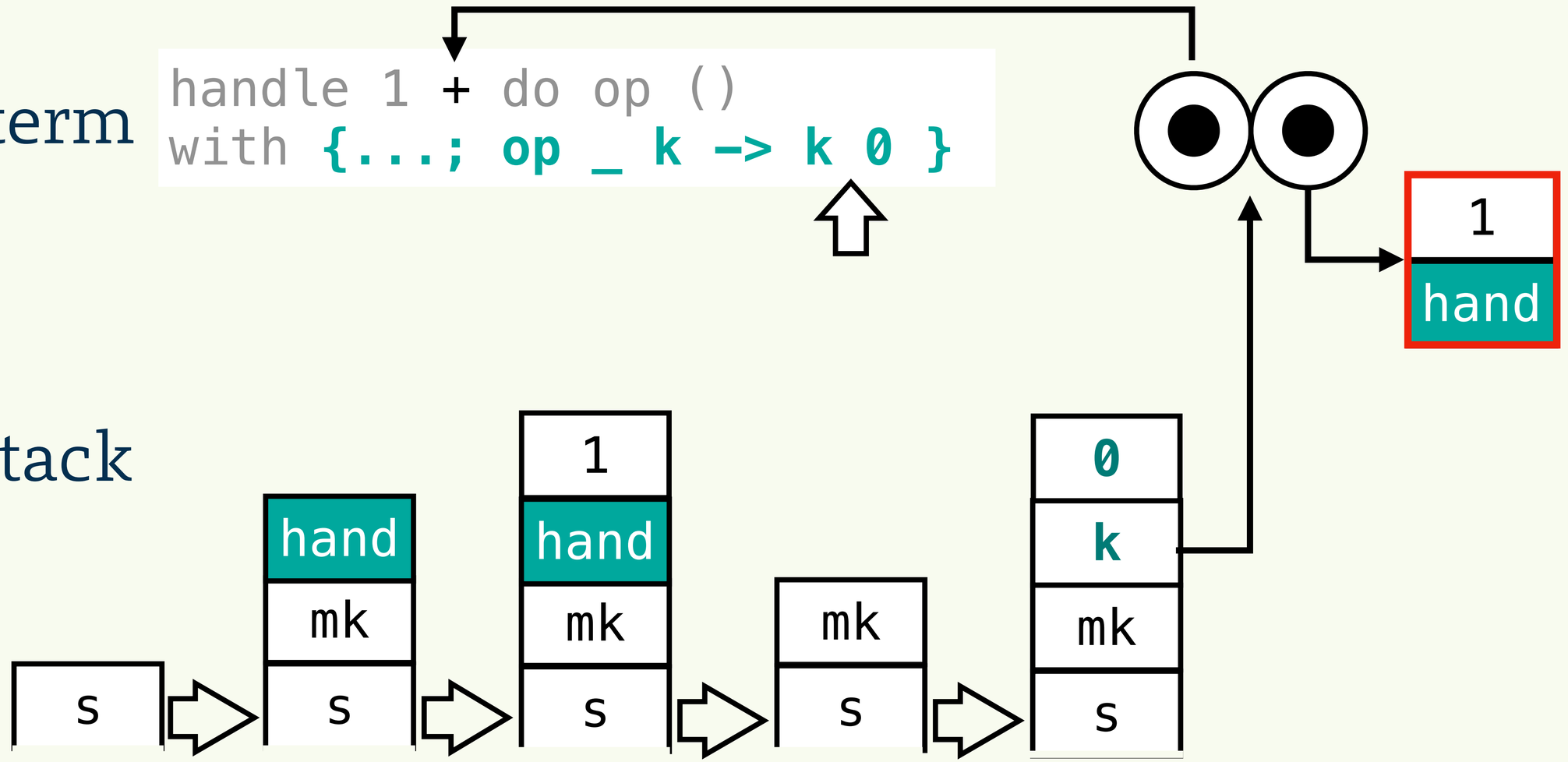


Effect Handlers in Target Language

Source term

```
handle 1 + do op ()
with {...; op _ k -> k 0 }
```

Target stack



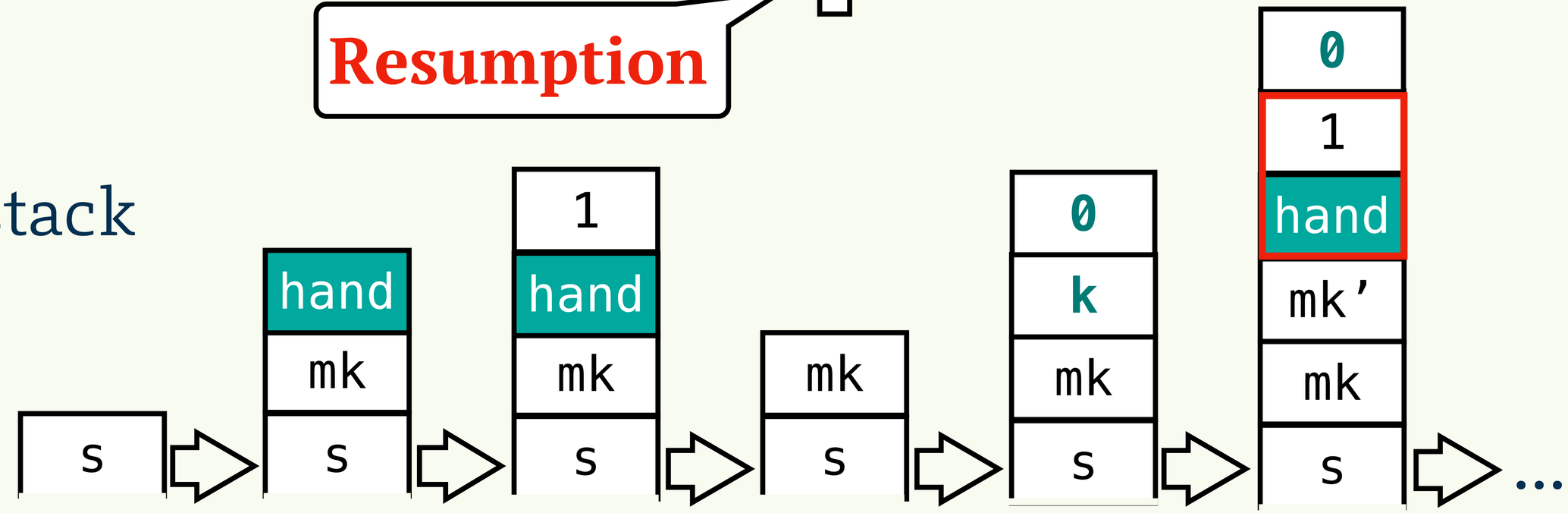
Effect Handlers in Target Language

Source term

```
handle 1 + do op ()
with {...; op _ k -> k 0 }
```

Resumption

Target stack



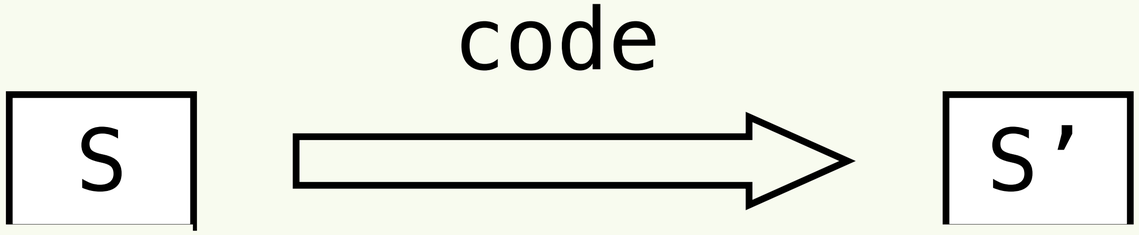
Typing Instructions in Target Language

Judgement

$$\Gamma \vdash \texttt{code} : S \Longrightarrow S'$$

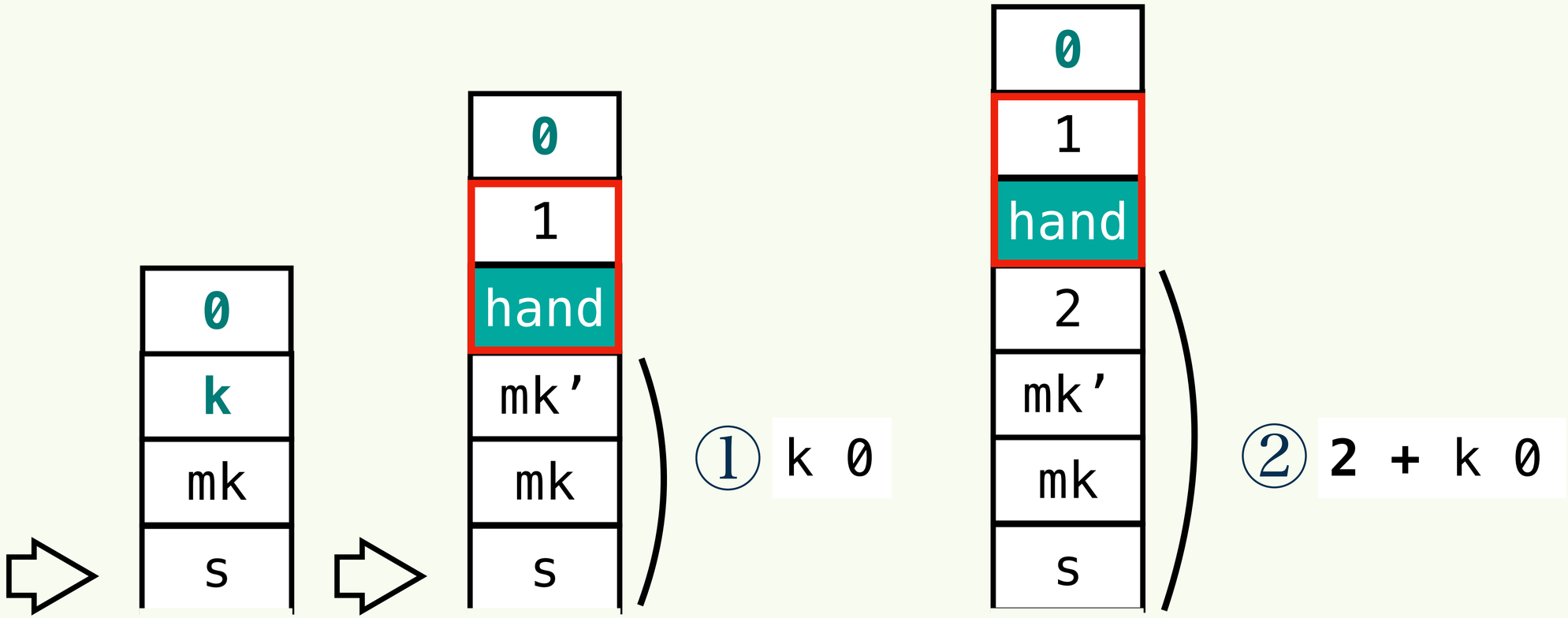
Before & after execution

Stack



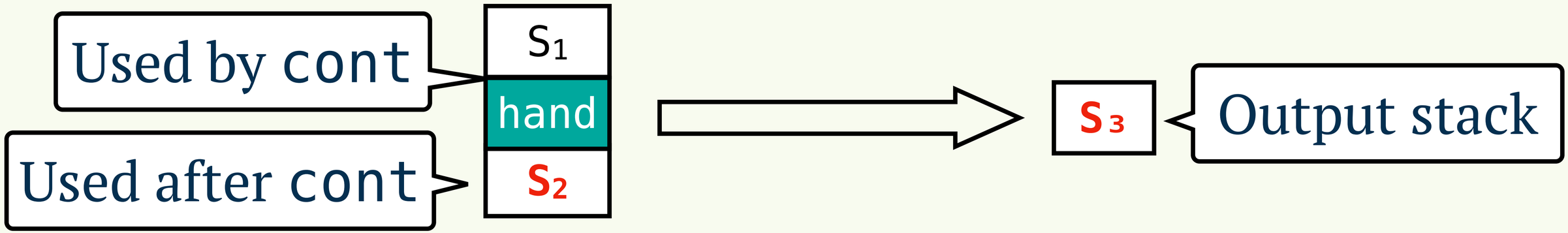
Motivation for Stack Polymorphism

Stacks before/after resumption are **not known** when capturing continuation



Typing Continuations with Stack Polymorphism

$$\Gamma \vdash \texttt{cont} : \forall S_2, S_3.\ (S_1 \texttt{ ++ Hand ... :: } S_2) \Longrightarrow S_3$$



Intrinsically Typed Compiler for Effect Handlers

Compiler for top-level program

```
compile : Cmp Γ (A , []) → Code Γ S (A :: S)
```

`Γ ⊢ M : (A , ε)`

`Γ ⊢ code : S ⟹ (A :: S)`

Type checked definition of function `compile`
= **Proof** of type preservation of compiler

Intrinsically Typed Compiler for Effect Handlers

Compiler for top-level program

```
compile : Cmp Γ (A , []) → Code Γ S (A :: S)
```

`Γ ⊢ M : (A , ε)`

`Γ ⊢ code : S ⟹ (A :: S)`

```
compileV : Val Γ A → Code Γ (ValTy A :: S) S' → Code Γ S S'
compileC : Cmp Γ (A , E) → PureCodeCont Γ (ValTy A :: S₁) (A' , E) → Code Γ (S₁ ++ HandTy Γ₁ S S' (A' , E) :: S) S'
-- auxiliary function for compiling handlers
compileH : Hdl Γ (C ⟹ D) → HandlerCode Γ C D
-- auxiliary function for compiling operation clauses
compileOps :
  OperationClauses Γ E₁ (B , E₂) →
  ∀{S₁ S₂ S₃ Γ₁ Γ'₁ A} → OperationCodes B E₁ E₂ Γ (ContTy Γ'₁ (ValTy B :: S₁) (A , E₂) :: (S₁ ++ HandTy Γ₁ S₂ S₃ (A , E₂) :: S₂)) S₃

compileV Unit = PUSH unit
compileV (Var x) = LOOKUP x
compileV {A = A ⇒ (B , E₁)} (Lam e) =
  ABS (λ {S₁ S₂ S₃ Γ₁ Γ'₁ A'} → compileC {S₁ = (ContTy Γ₁ (ValTy B :: S₁) (A' , E₁)) :: _} e RET)

compileC (Handle e With h) k = MARK (compileH h) k (compileC {S₁ = []} e UNMARK)
compileC (Let_In_ {A = A}{E = E₁}{B = B} e1 e2) k =
  compileC e1 $ BIND (compileC {S₁ = (ContTy _ (ValTy B :: _) (_ , _) :: _)} e2 RET) k
compileC (Return v) k = compileV v k
compileC (Do l v) k = compileV v $ CALLOP l k
compileC (App v1 v2) k = compileV v1 $ compileV v2 $ APP k

compileH {D = (B , E₂)} (ℵx ret |ℵx,r ops) {Γ₁} {Γ'₁} {S₁} {S₂} {S₃} {A'} =
  (compileC {S₁ = ContTy Γ'₁ (ValTy B :: S₁) (A' , E₂) :: _} ret RET , compileOps ops)

compileOps {E₁ = []} [] = []
compileOps {E₁ = (op A' B') :: E'}{B = B}{E₂ = E₂} (e :: es) {S₁} {S₂} {S₃} {Γ₁} {Γ'₁} {A₁} =
  (compileC {S₁ = ContTy Γ'₁ (ValTy B :: S₁) (A₁ , E₂) :: _} e RET) :: (compileOps es)

compile : Cmp Γ (A , []) → Code Γ S (ValTy A :: S)
compile c = INITHAND (compileC {S₁ = []} c UNMARK)
```

Definition type checked

All Done

Future Work

- Semantics preservation
  - Extending compiler calculation [Pickard+'21]
- Other continuation facilities
  - shift0/reset0 , shallow handlers , etc
- Optimization based on continuation usage
  - Extend with multiplicities[Atkey'18]

Summary

This work

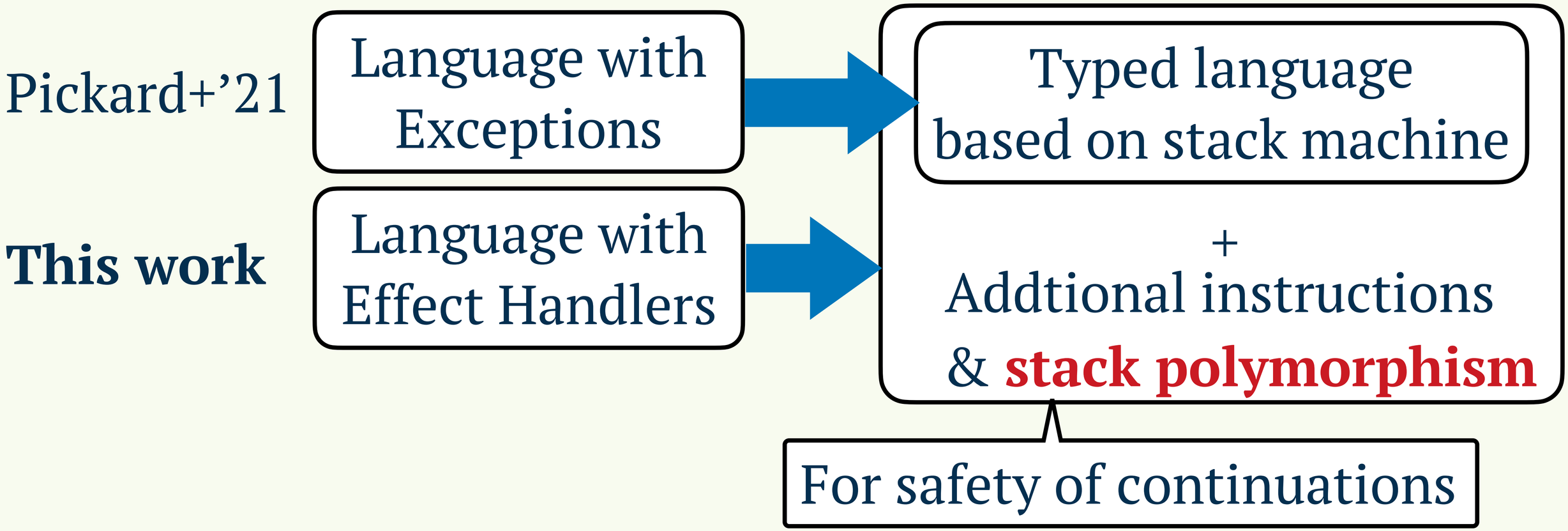
An intrinsically typed compiler for effect handlers in Agda
(Using stack polymorphism to type continuations)

Future work

- Semantics preservation
- Other continuation facilities
- Optimization based on continuation usage

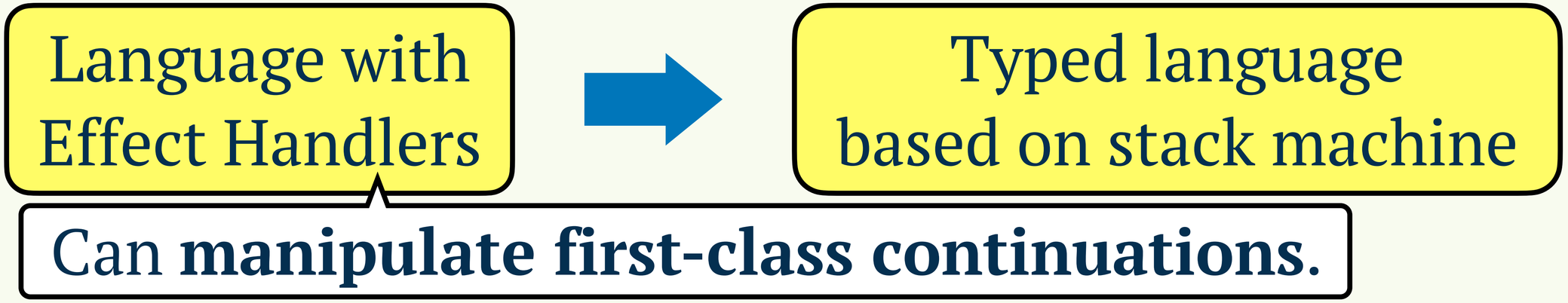
Summary

An intrinsically typed compiler for effect handlers in Agda



Out Contributions

Implementing the intrinsically typed compiler
to prove type-preservation of compilation



Ensure safety of continuation capture and resumption

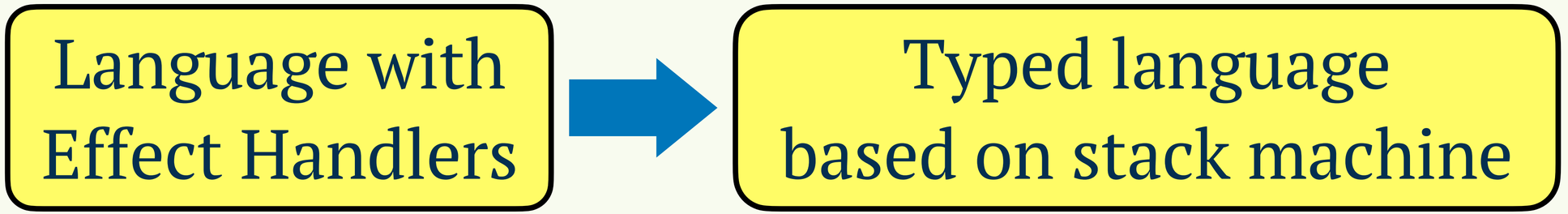
Incorporating **stack polymorphism**[Morrisett+'02]
to

Novelty : Using in the context of compiling continuations.

Background / Type-Preserving Compilation

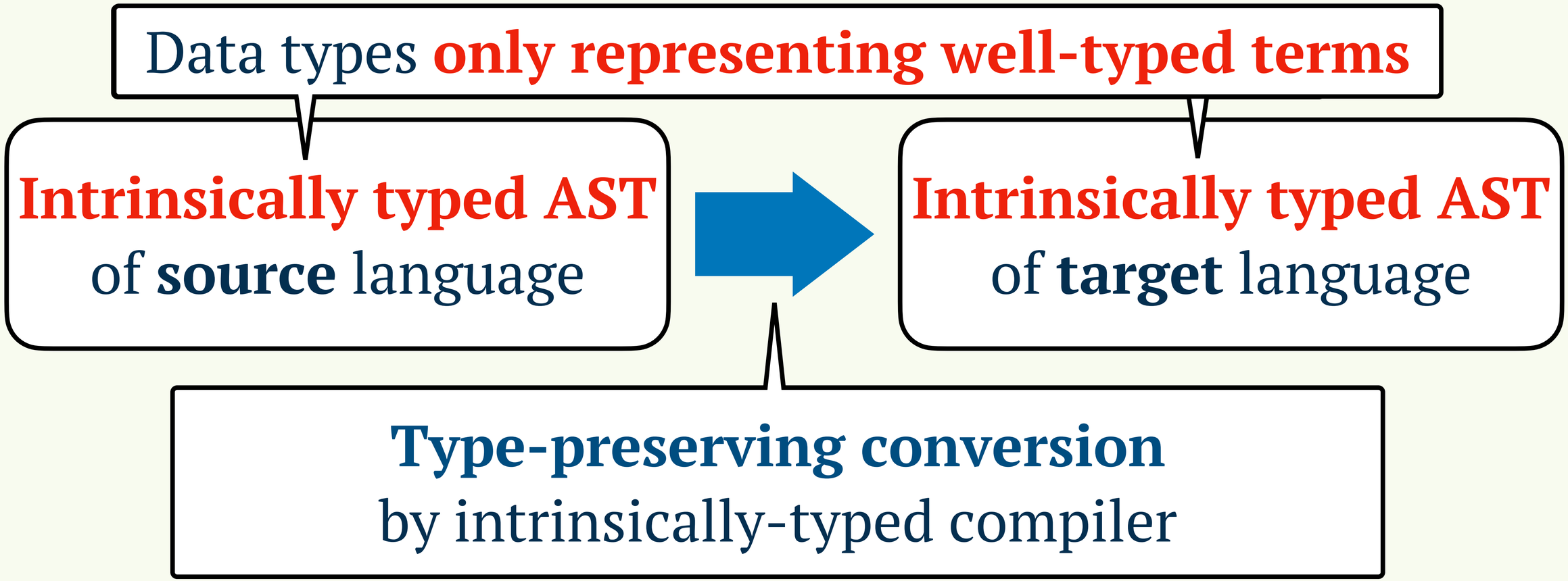
Morrisett'99

Our work



Background / Intrinsically Typed Compiler[Pickard+'21]

An approach to prove type-preserving compilation

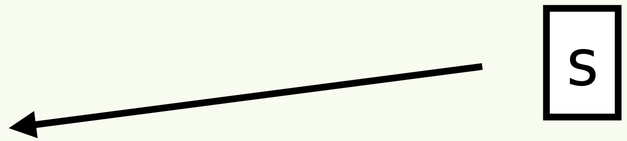


Effect Handlers in Target Language

Source term

```
handle 1 + (do op ())
with {...; op _ k -> k 0 }
```

Target Stack



Effect Handlers in Target Language

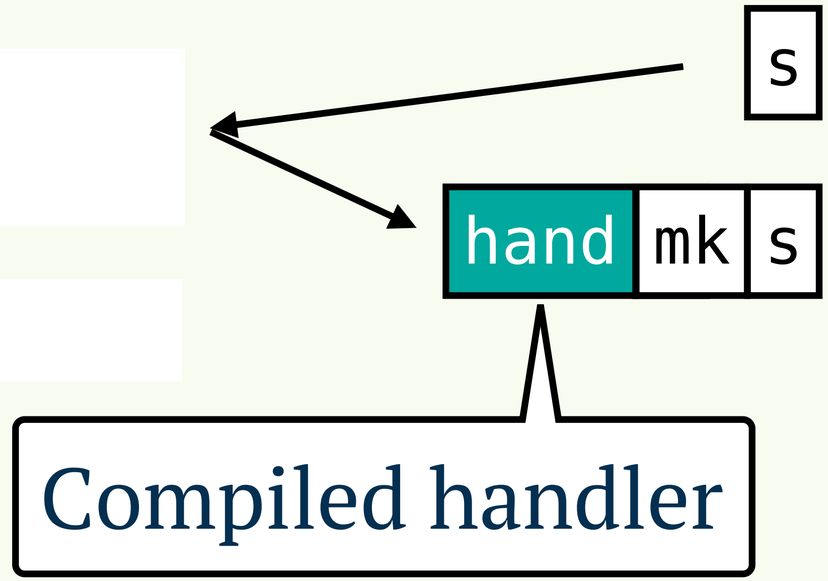
Source term

```
handle 1 + (do op ())
with {...; op _ k -> k 0 }
```

⇨

```
1 + (do op ())
```

Target Stack



Effect Handlers in Target Language

Source term

```
handle 1 + (do op ())
with {...; op _ k -> k 0 }
```

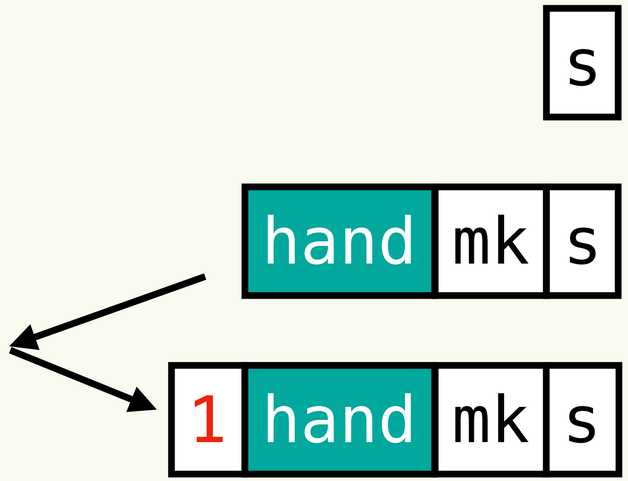
⇨

```
1 + (do op ())
```

⇨

```
1 + (do op ())
```

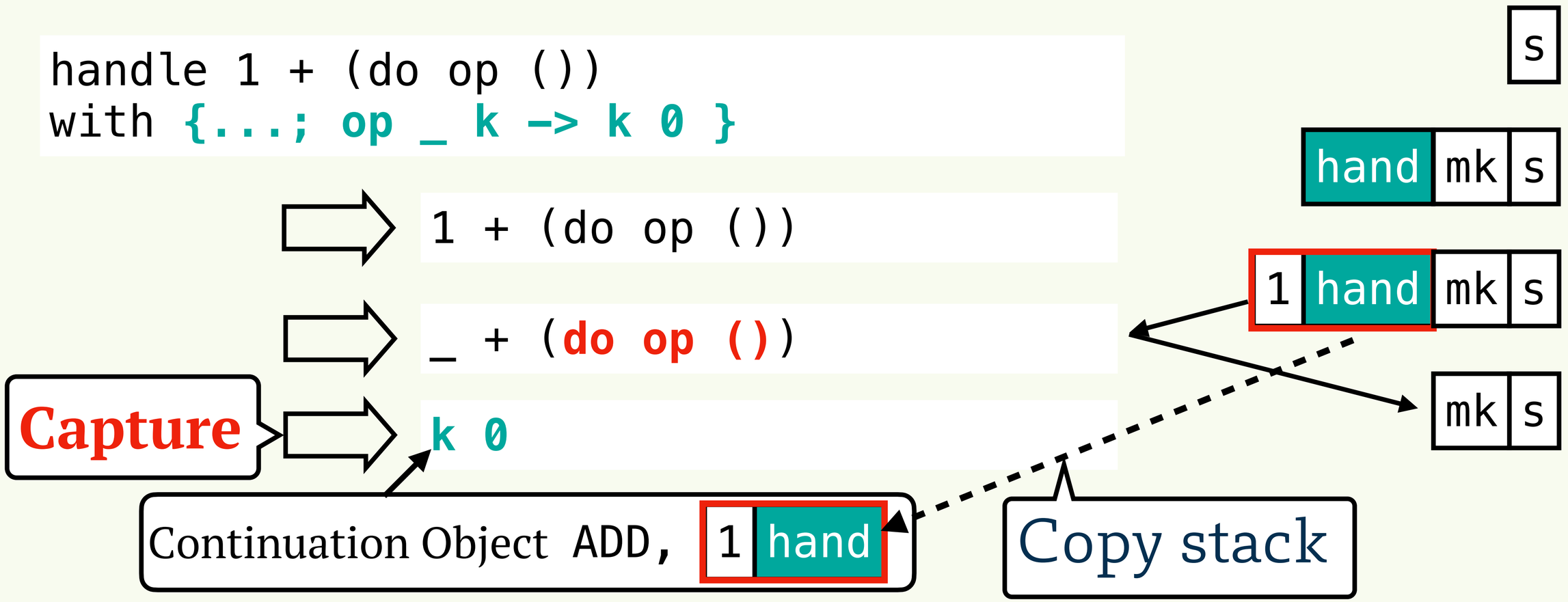
Target Stack



Effect Handlers in Target Language

Source term

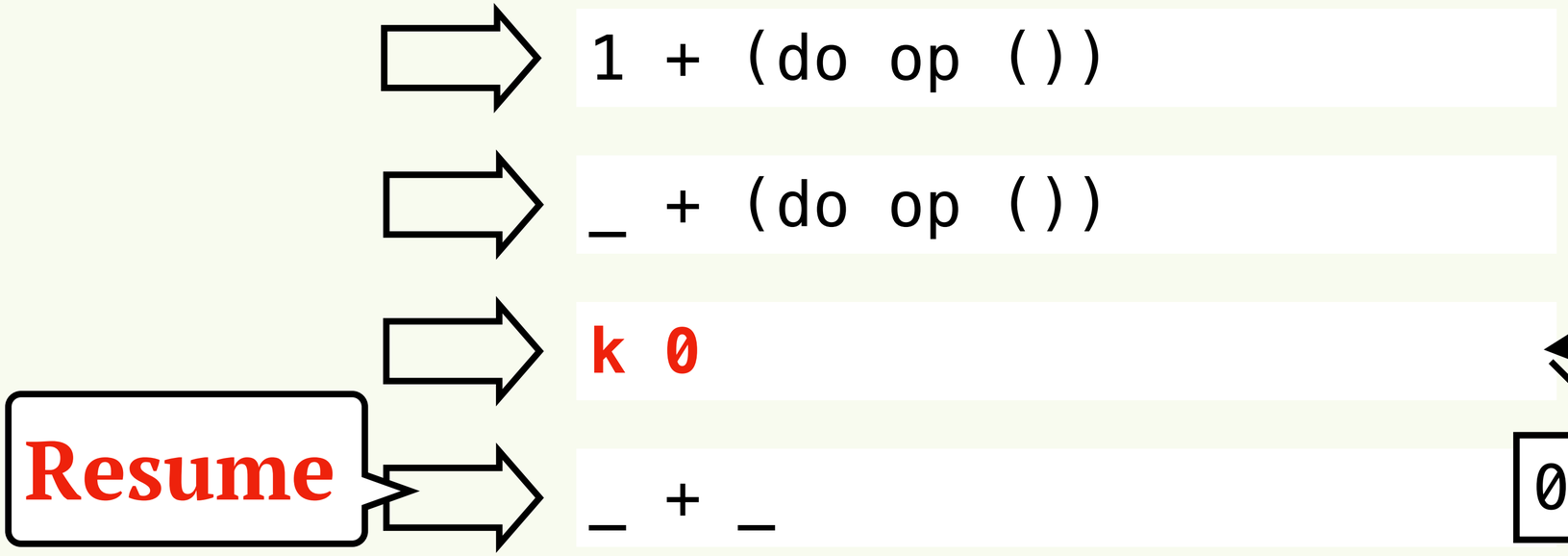
Target Stack



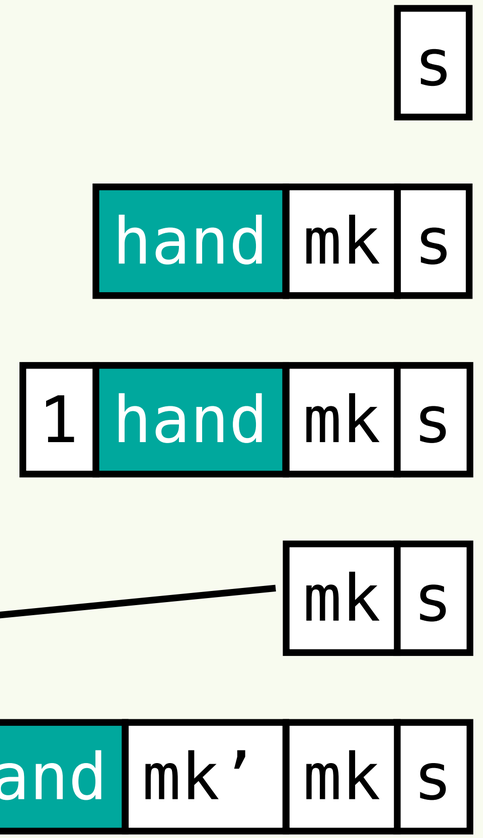
Effect Handlers in Target Language

Source term

```
handle 1 + (do op ())
with {...; op _ k -> k 0 }
```



Target Stack

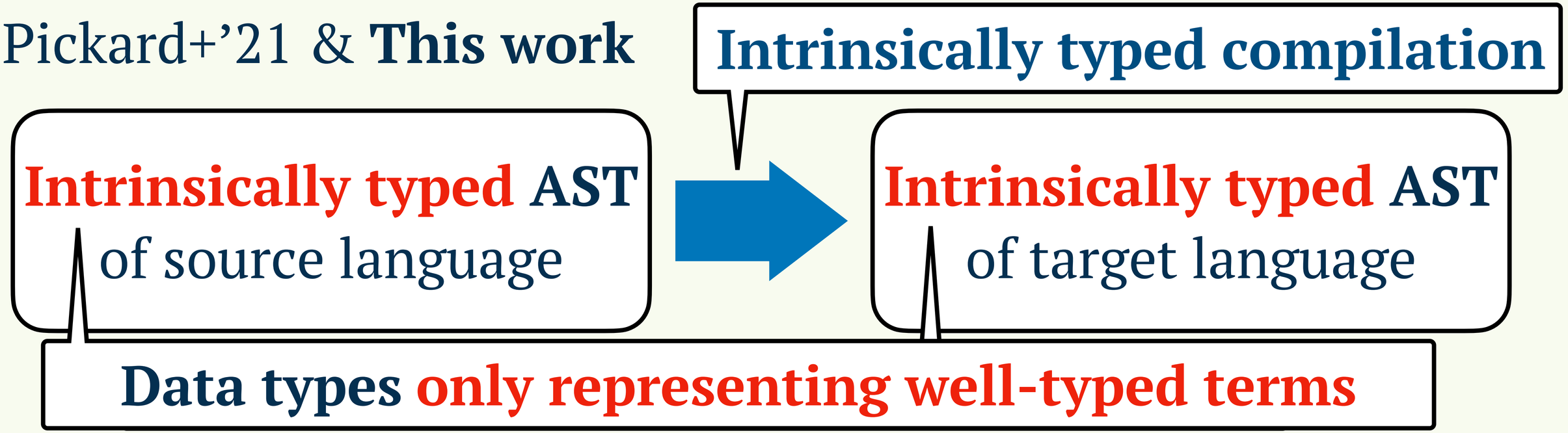


Background / Intrinsically Typed Compiler[Rouvoet +'21]

An approach to prove type-preserving compilation

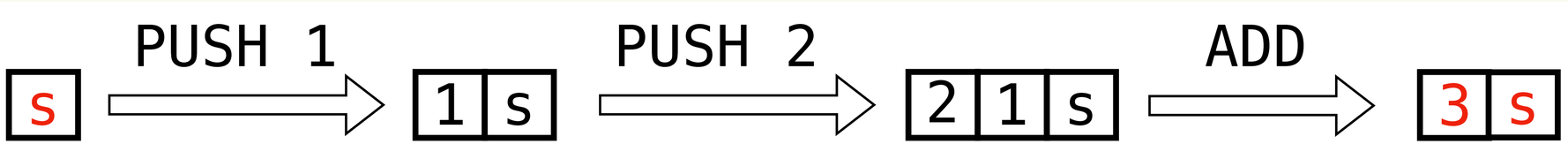
Type checked definition = Proof of type preservation

Pickard+'21 & **This work**



Target Language L_T : Language based on stack machine

ex. `PUSH 1 (PUSH 2 (ADD HALT))` : **S ⟹ N :: S**



Intrinsically-typed AST

```
data Code (Γ : Ctx) : StackTy → StackTy → Set where
  PUSH : Val A → Code Γ (A :: S) S' → Code Γ S S'
```

Rest instructions

Γ ⊢ inst : S ⟹ S'

Source Language L_s / Intrisically typed AST

Data types only representing well-typed terms in L_s

```
App (Lam ...) unit : Cmp [] (A , E)
```

❌ `App 0 0`

```
data Val (Γ : Ctx) : VTy → Set
data Cmp (Γ : Ctx) : CTy → Set
```

`Γ ⊢ V : A`

`Γ ⊢ M : (A , E)`

```
data Cmp Γ where
  App : Val Γ (A ⇒ C) → Val Γ A → Cmp Γ C
  ...
```

$$\frac{\Gamma \vdash v : A \Rightarrow C \qquad \Gamma \vdash w : A}{\Gamma \vdash v\ w : C}\ [\text{T-APP}]$$

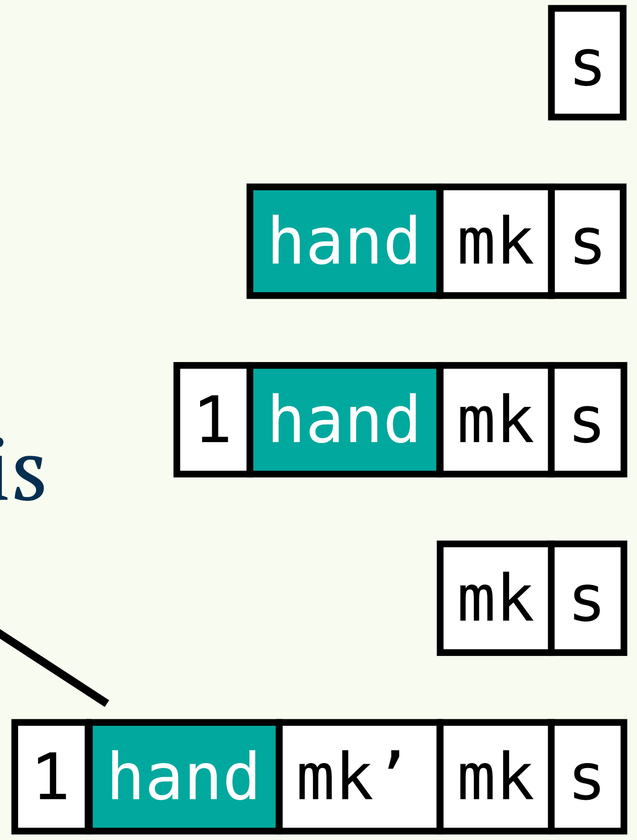
Motivation for Stack Polymorphism

Source term

```
handle 1 + (do op ())
with {...; op _ k -> k 0 }
```

When **compiling** the continuation,
the stack required by resumption handler is
unknown

Target stack



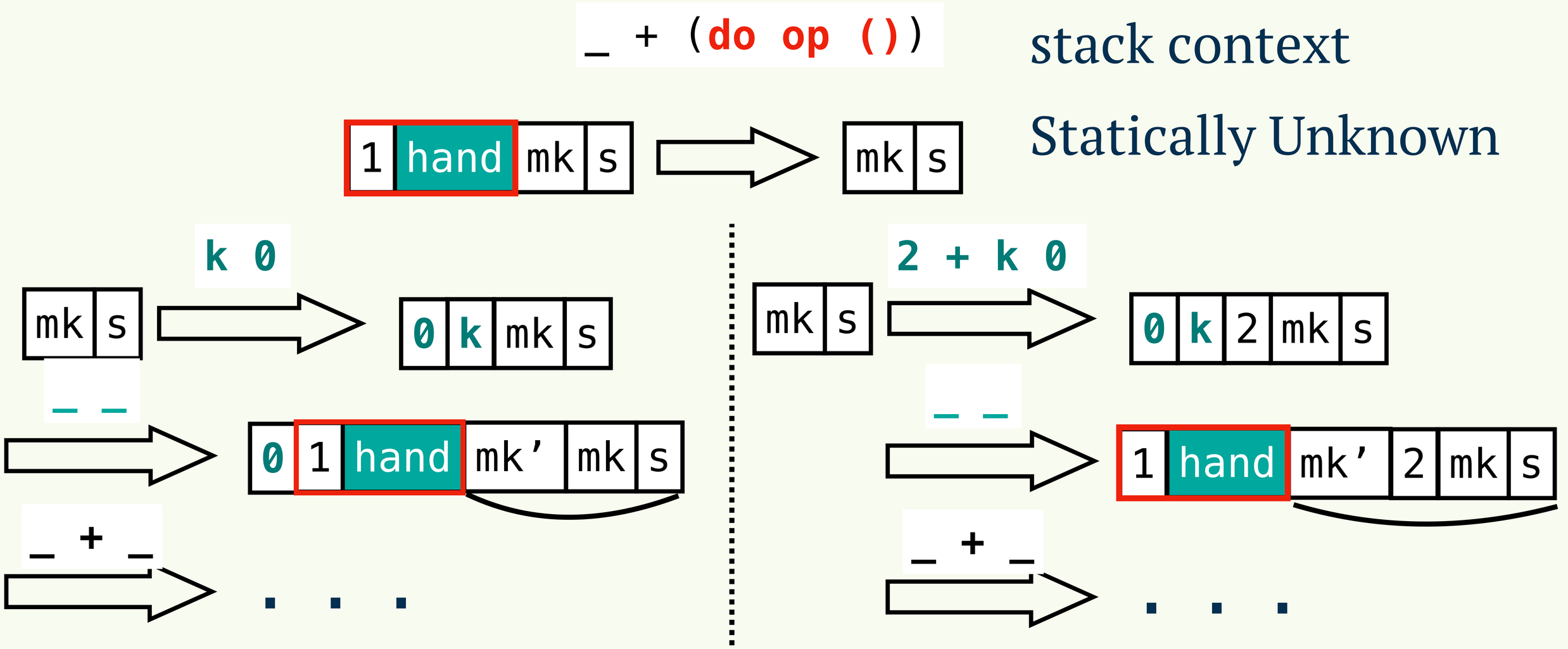
Type of continaution in

```
PureCodeCont Γ S₁ C =
  ∀{S₂ S₃} → Code Γ (S₁ ++ HandTy Γ₁ S₂ S₃ C :: S₂) S₃
```

Continuation can be resumed with arbitrary stacks

```
Γ ⊢ inst : ∀S₂ S₃. ⟹ (A :: S)
```

Motivation for Stack Polymorphism



Intrinsically Typed Compiler / Keypoint

Previous CPS compiler for exception language[Pickard +'21]

```
compileC :
    Cmp Γ (A , E) →
     Code ... →
    Code (S₁ ++ HandTy S S' :: S) S'
```

CPS Compiler for effectful computation

```
compileC :
  Cmp Γ (A , E) →
  (∀{S S'} → Code ... ) →
  Code Γ (S₁ ++ HandTy Γ₁ S S' (A' , E) :: S) S'
```

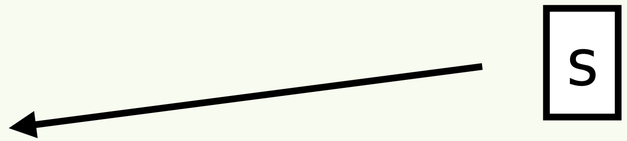
Redefined for first-class continuation

Effect Handlers in Target Language

Source term

```
handle 1 + (do op ())
with {...; op _ k -> k 0 }
```

Target Stack



Effect Handlers in Target Language

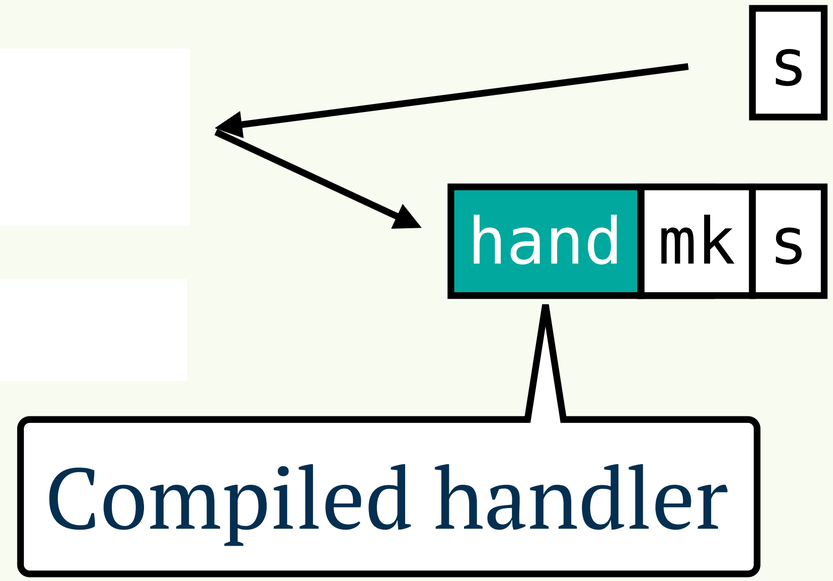
Source term

```
handle 1 + (do op ())
with {...; op _ k -> k 0 }
```

⇨

```
1 + (do op ())
```

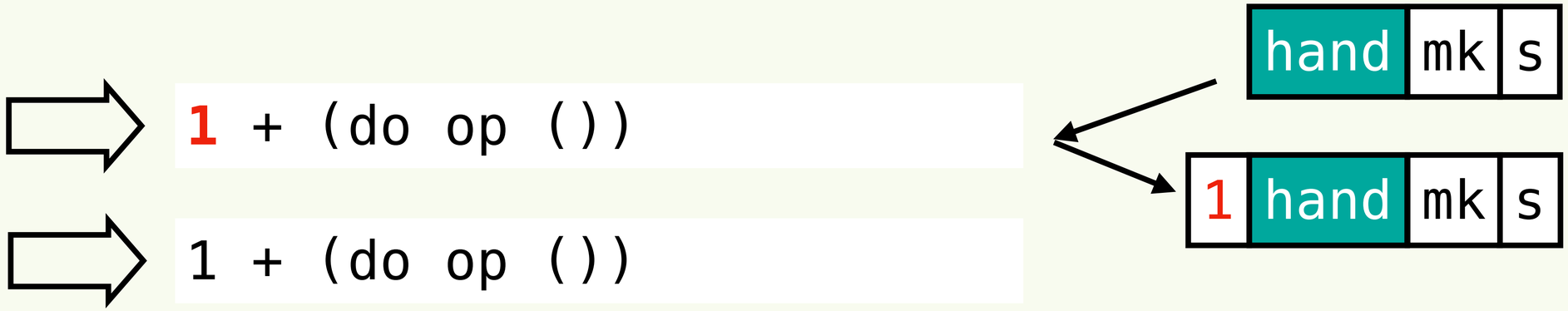
Target Stack



Effect Handlers in Target Language

Source term

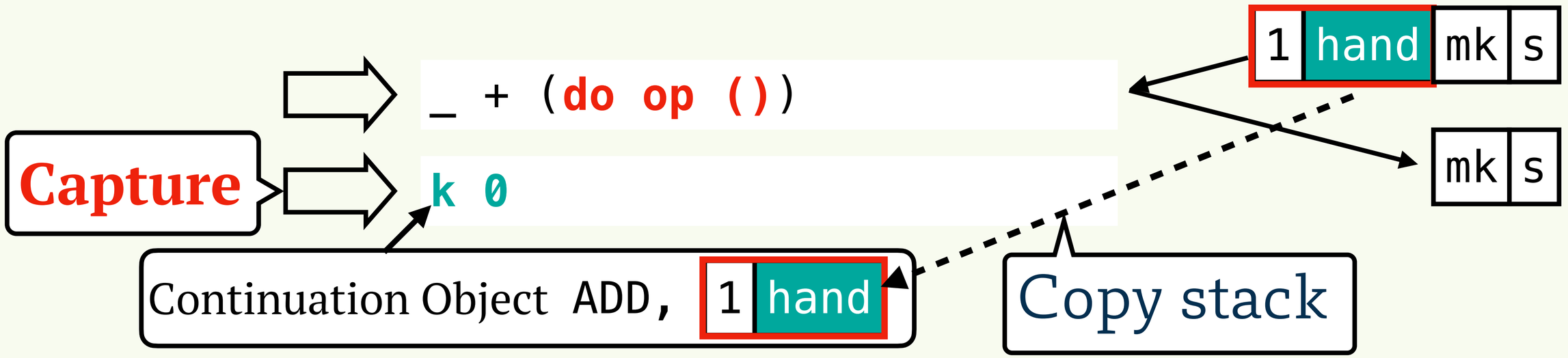
Target Stack



Effect Handlers in Target Language

Source term

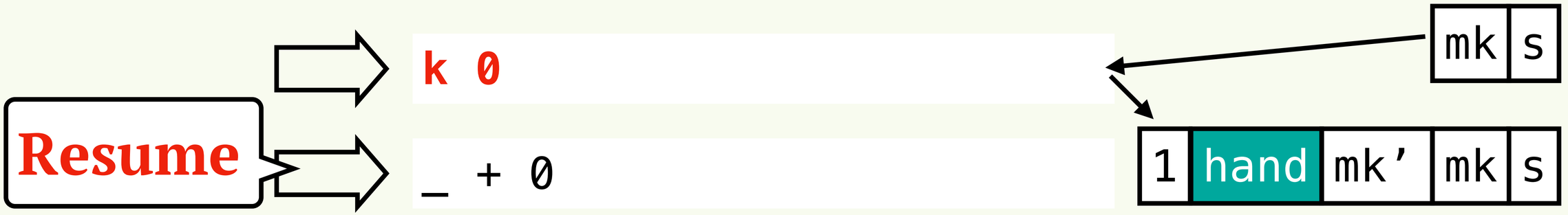
Target Stack



Effect Handlers in Target Language

Source term

Target Stack

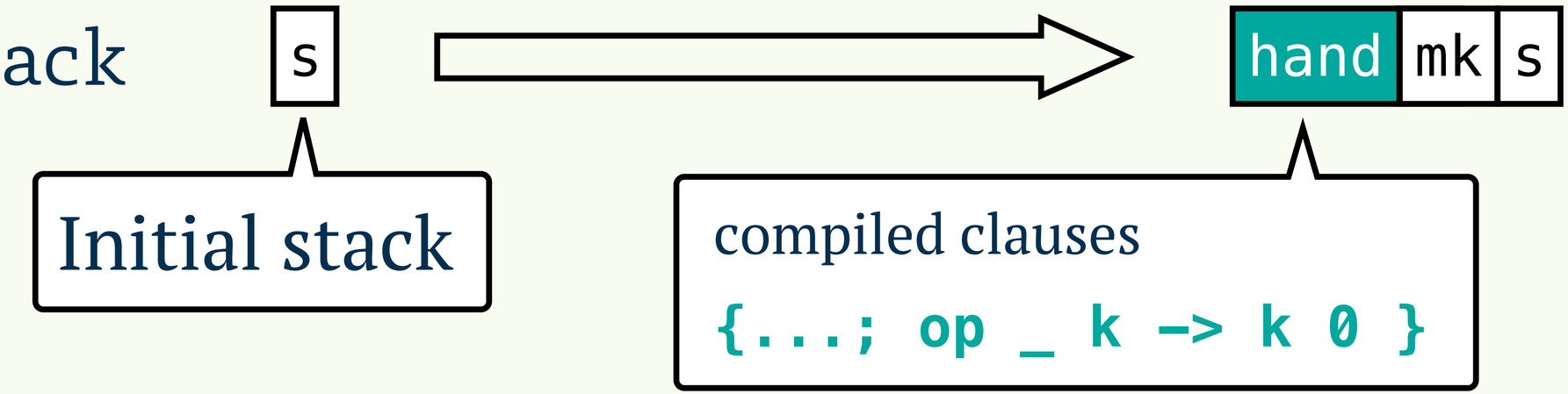


Effect Handlers in Target Language

Source term

```
handle 1 + do op ()
with {...; op _ k -> k 0 }
```

Target stack

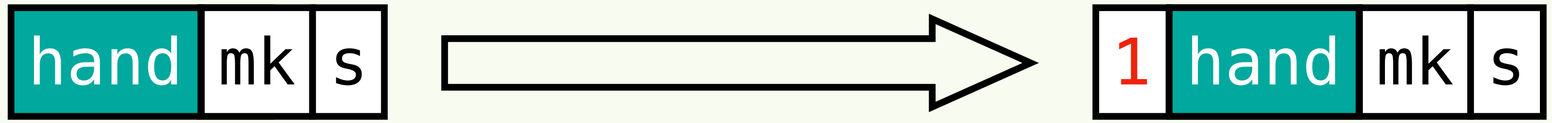


Effect Handlers in Target Language

Source term

```
1 + (do op ())
```

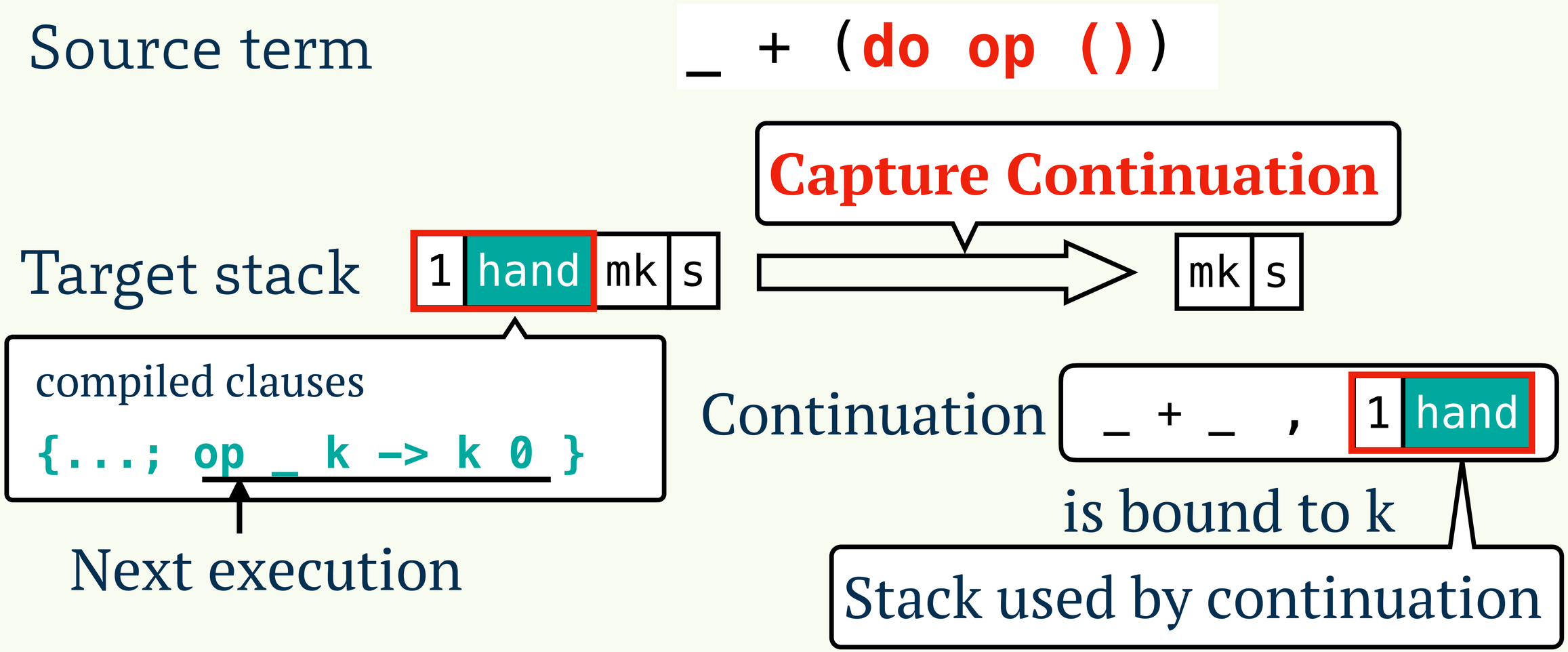
Target stack



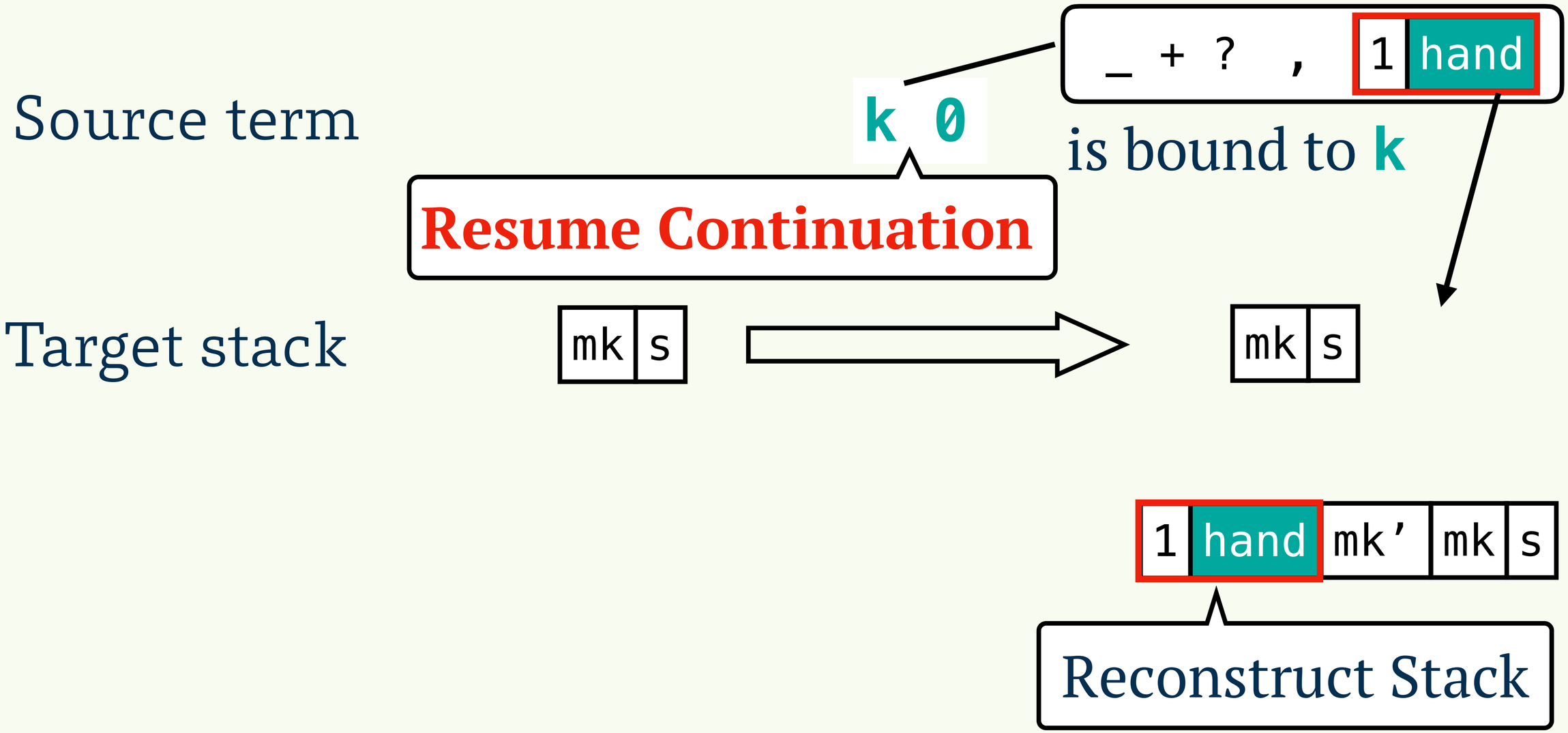
compiled clauses

```
{...; op _ k -> k 0 }
```

Effect Handlers in Target Language



Effect Handlers in Target Language

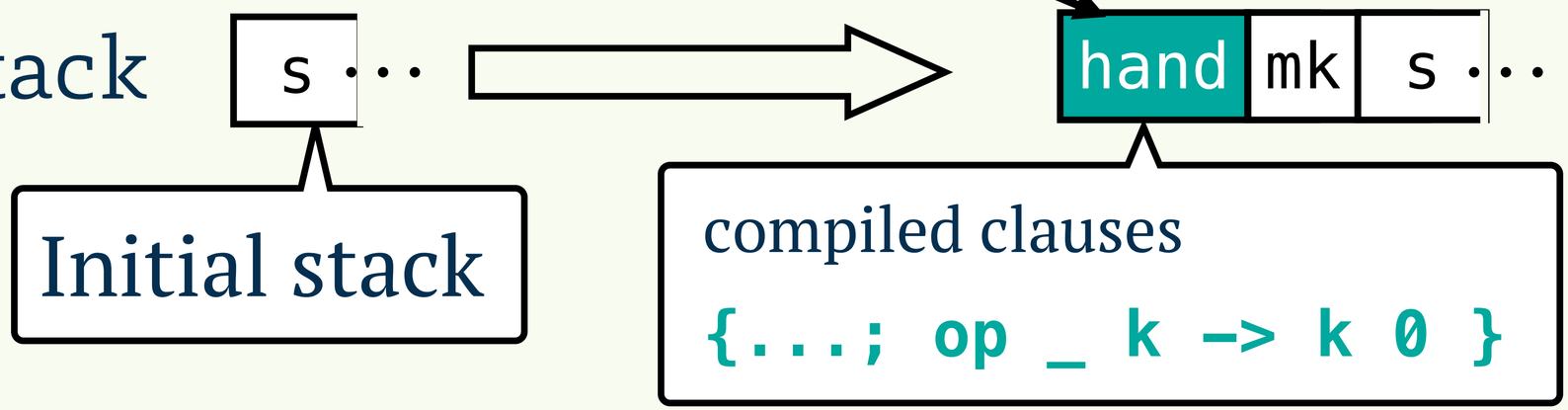


Effect Handlers in Target Language

Source term

```
handle 1 + do op ()
with {...; op _ k -> k 0 }
```

Target stack



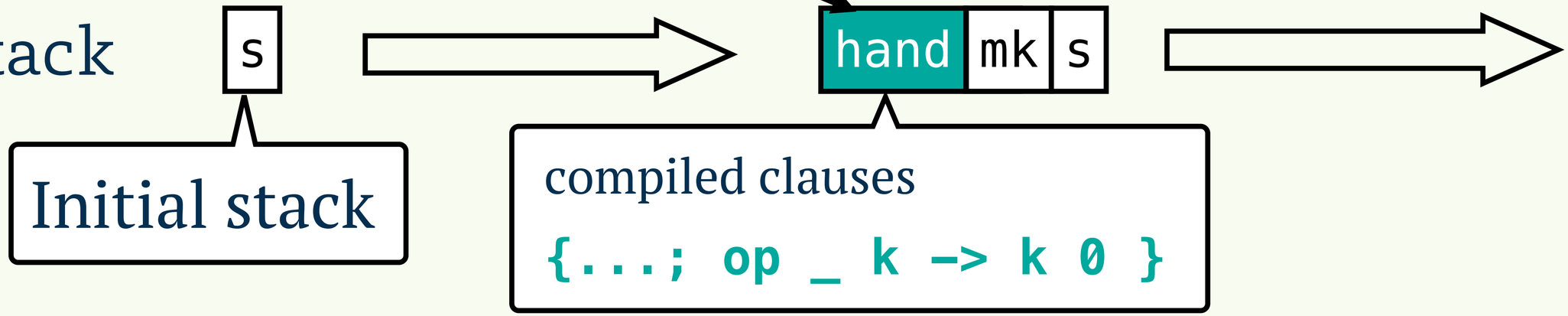
Effect Handlers in Target Language

Source term

```
handle 1 + do op ()
with {...; op _ k -> k 0 }
```

```
1 + (do op ())
```

Target stack



Effect Handlers in Target Language

Source term

```
1 + (do op ())
```

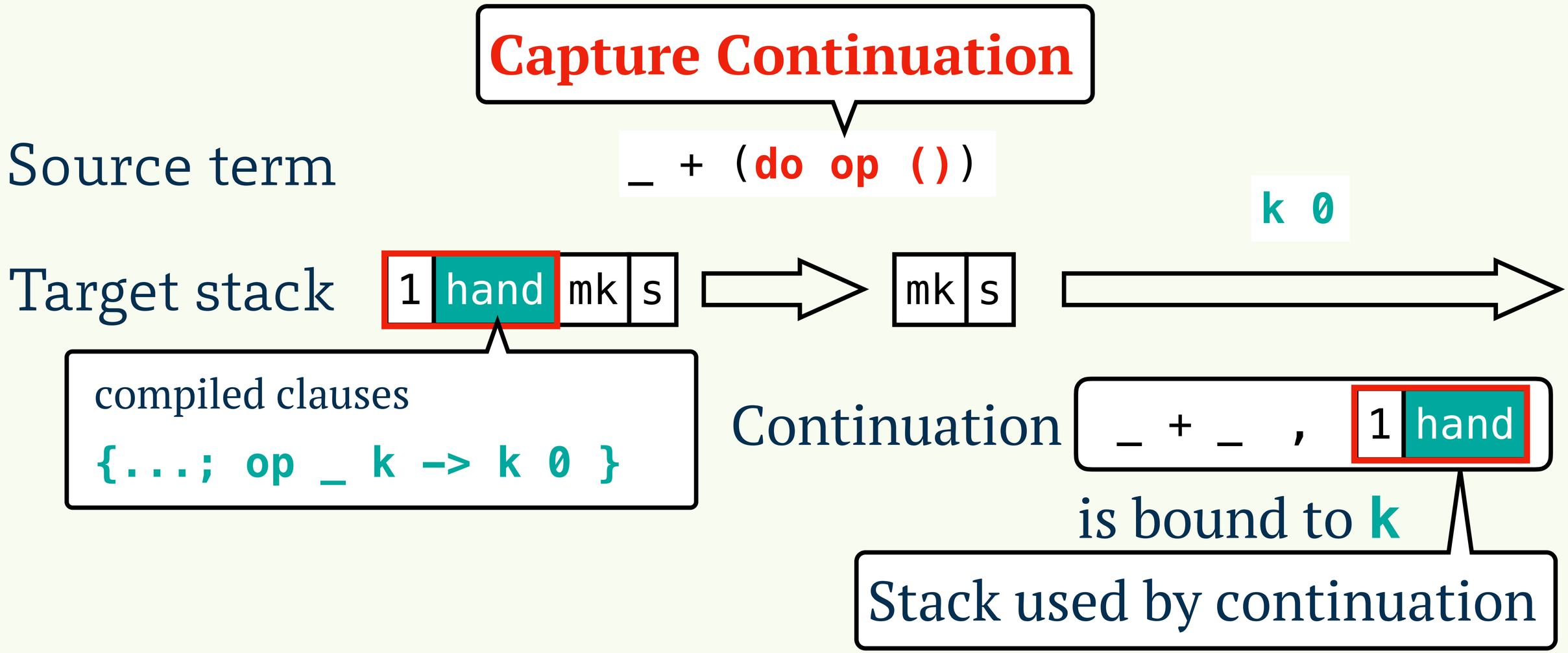
```
_ + (do op ())
```

Target stack

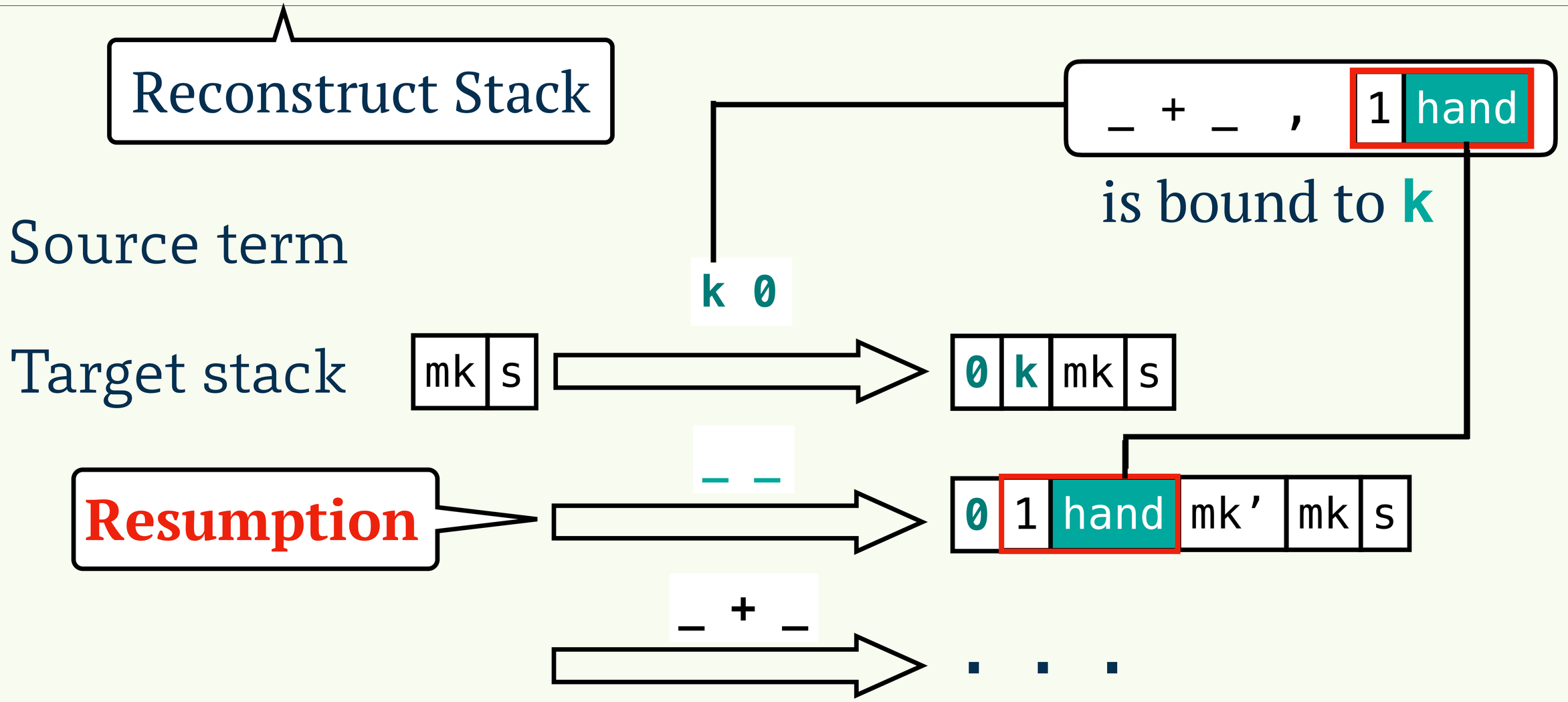
compiled clauses

```
{...; op _ k -> k 0 }
```

Effect Handlers in Target Language



Effect Handlers in Target Language

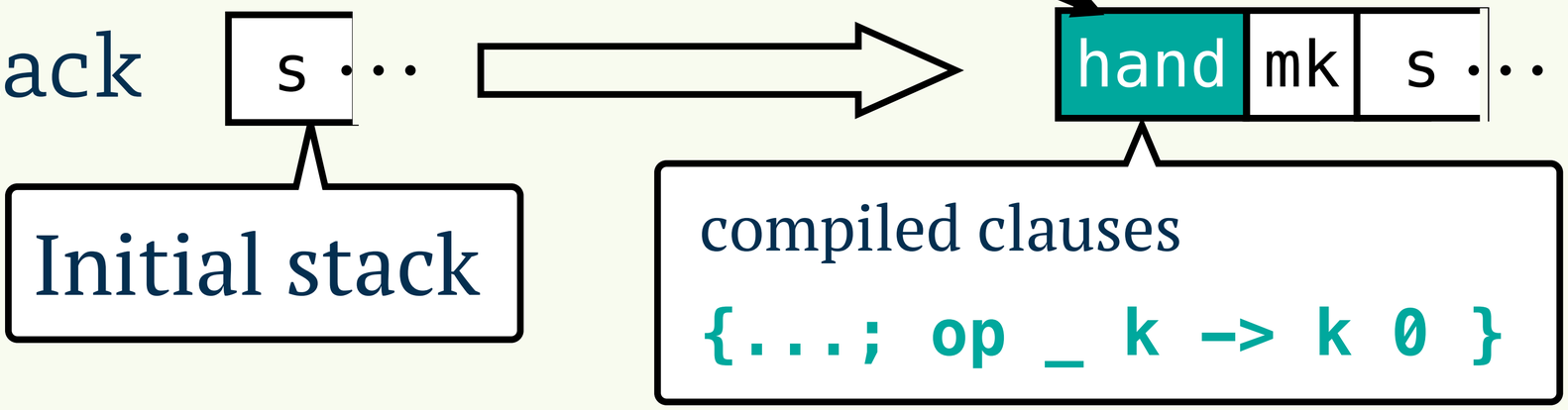


Effect Handlers in Target Language

Source term

```
handle 1 + do op ()
with {...; op _ k -> k 0 }
```

Target stack

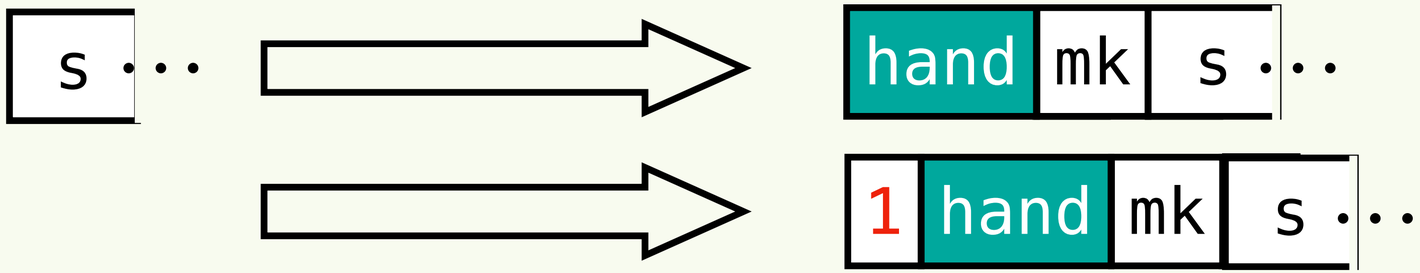


Effect Handlers in Target Language

Source term

```
handle 1 + do op ()
with {...; op _ k -> k 0 }
```

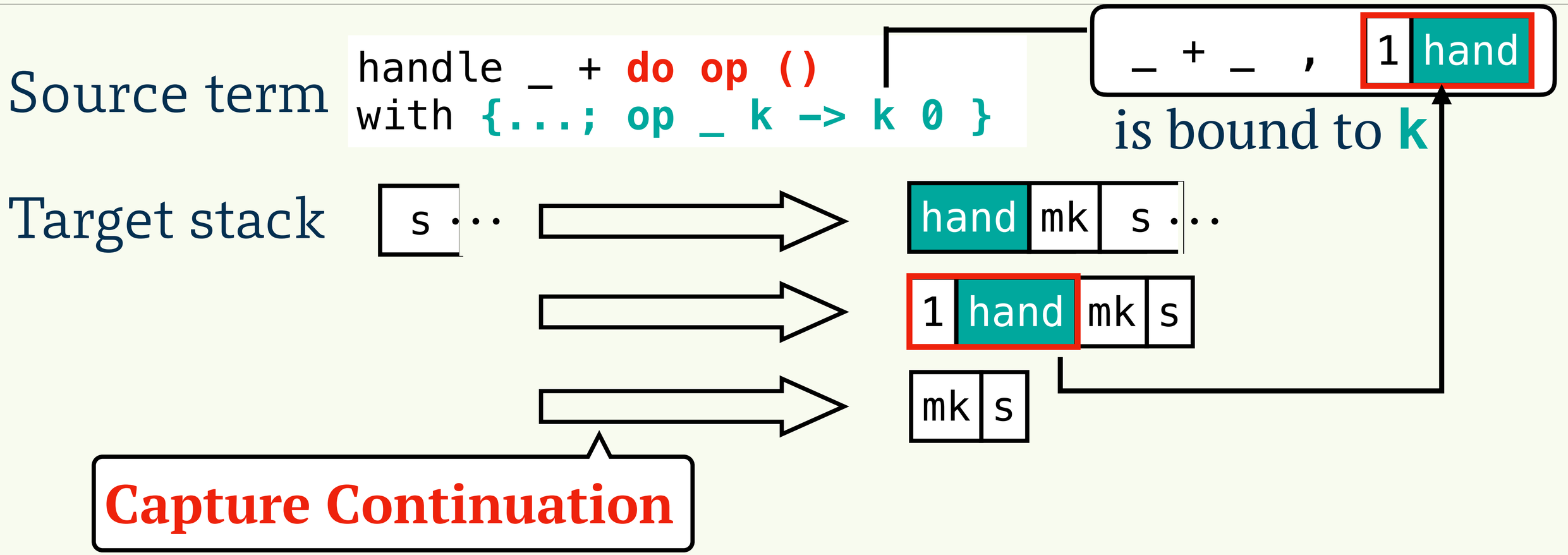
Target stack



Effect Handlers in Target Language

Source term

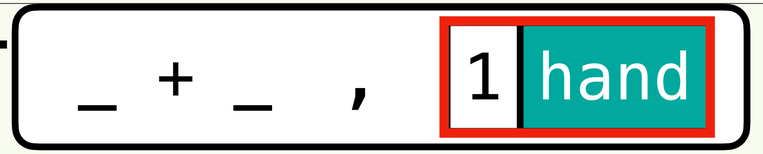
```
handle _ + do op ()
with {...; op _ k -> k 0 }
```



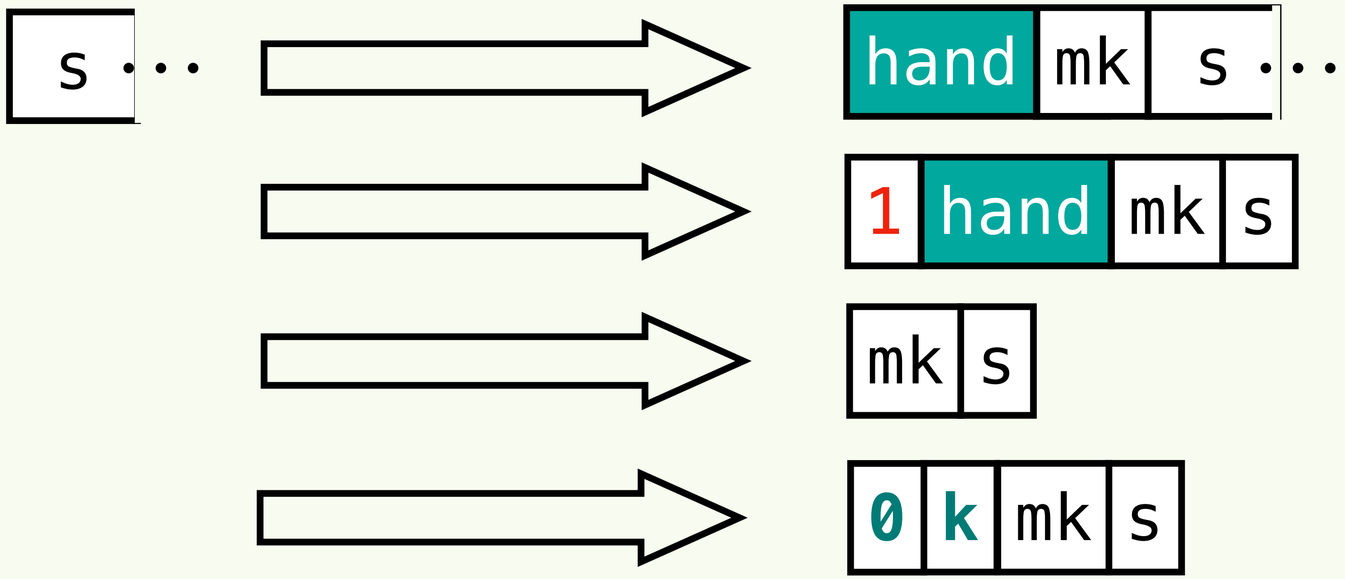
Effect Handlers in Target Language

Source term

```
handle _ + _
with {...; op _ k -> k 0 }
```



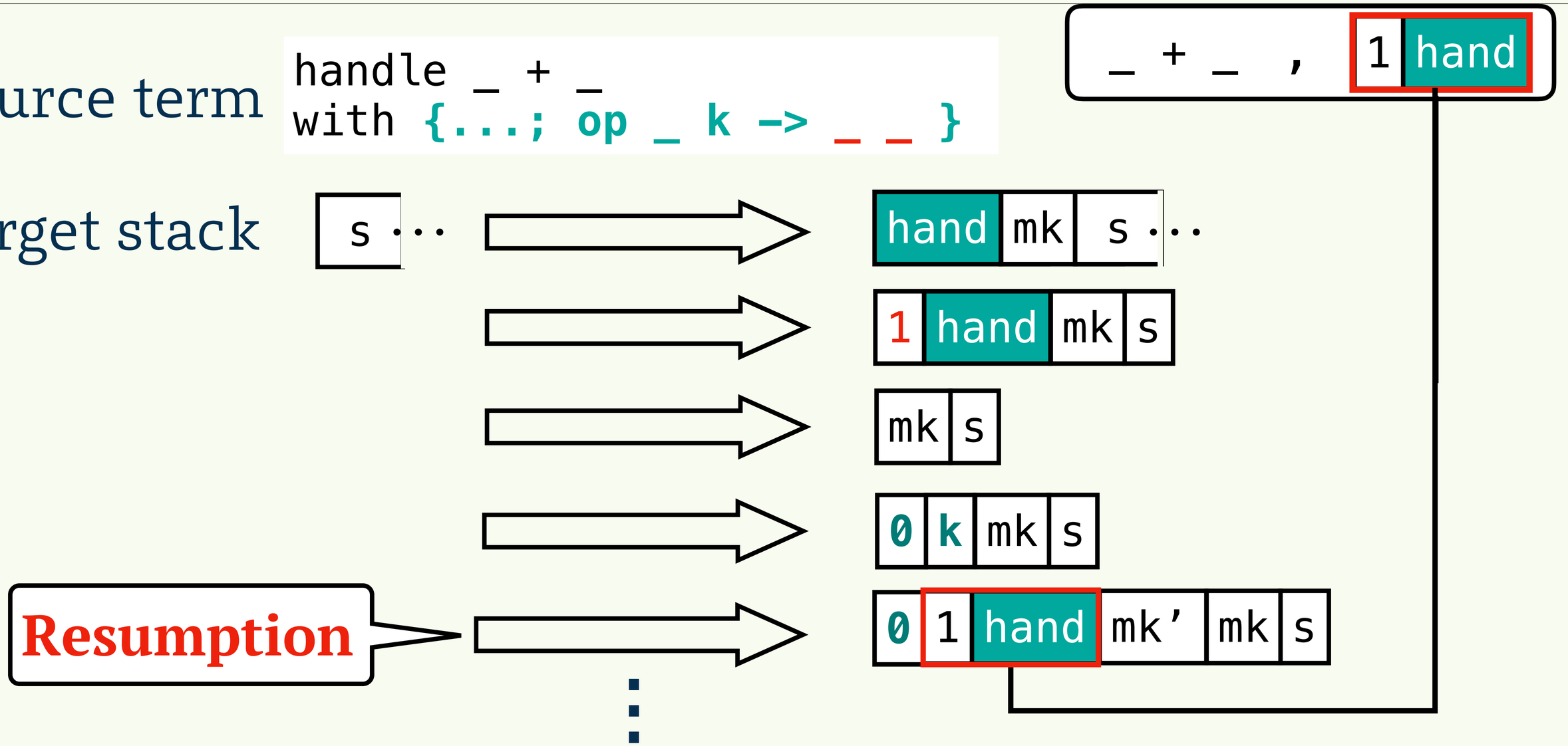
Target stack



Effect Handlers in Target Language

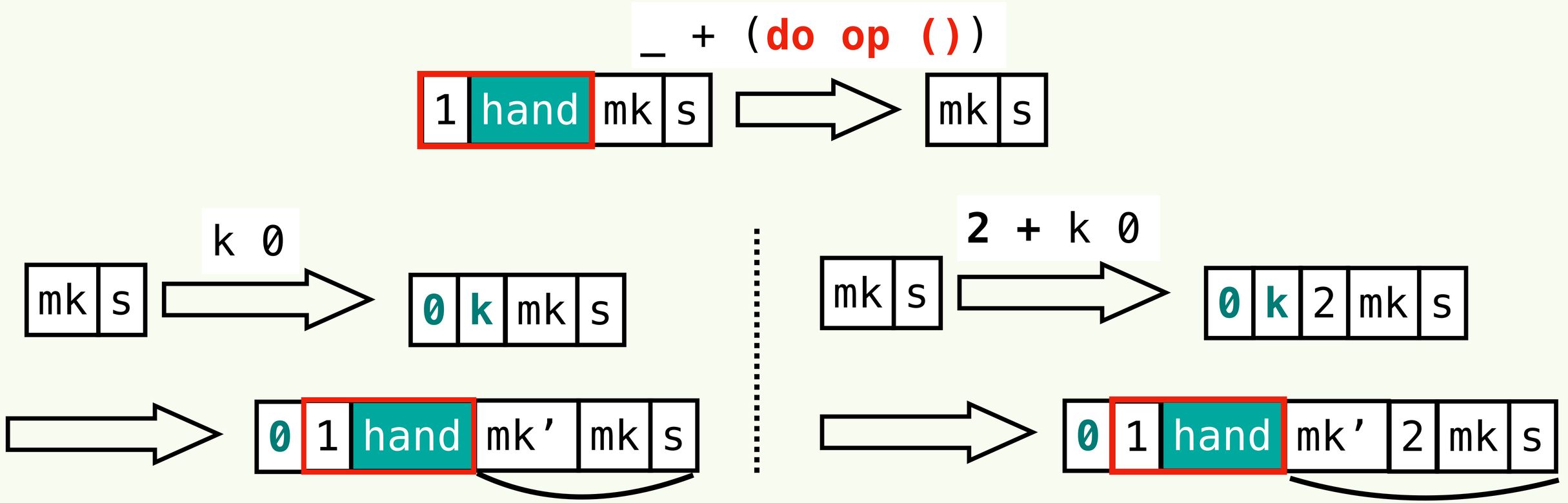
Source term

```
handle _ + _
with {...; op _ k -> _ _ }
```



Motivation for Stack Polymorphism

Stack context when captiring continuation at calling op



Stack context at resumption is **Statically Unknown**

Effect Handlers in Target Language

Source term

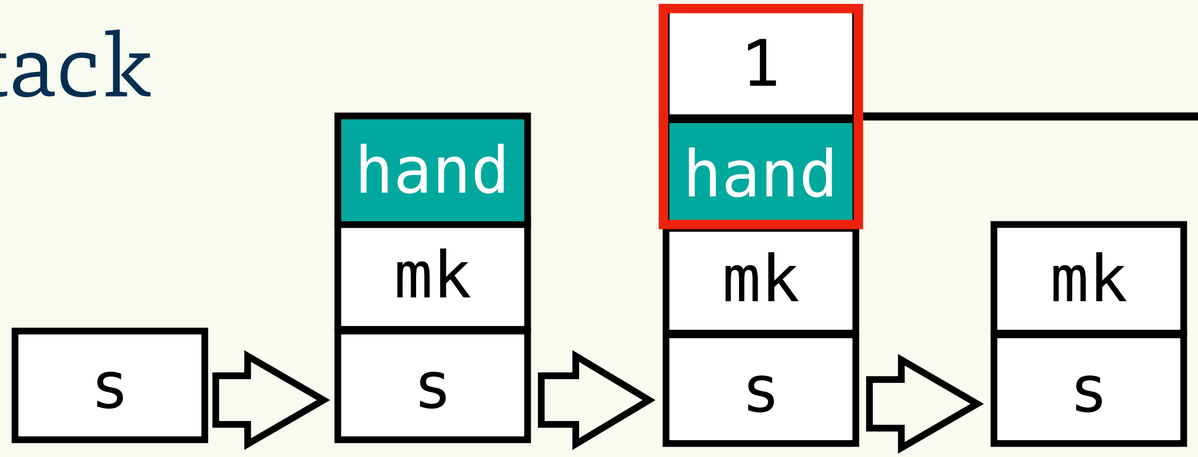
```
handle _ + do op ()
with {...; op _ k -> k 0 }
```

Capture Continuation

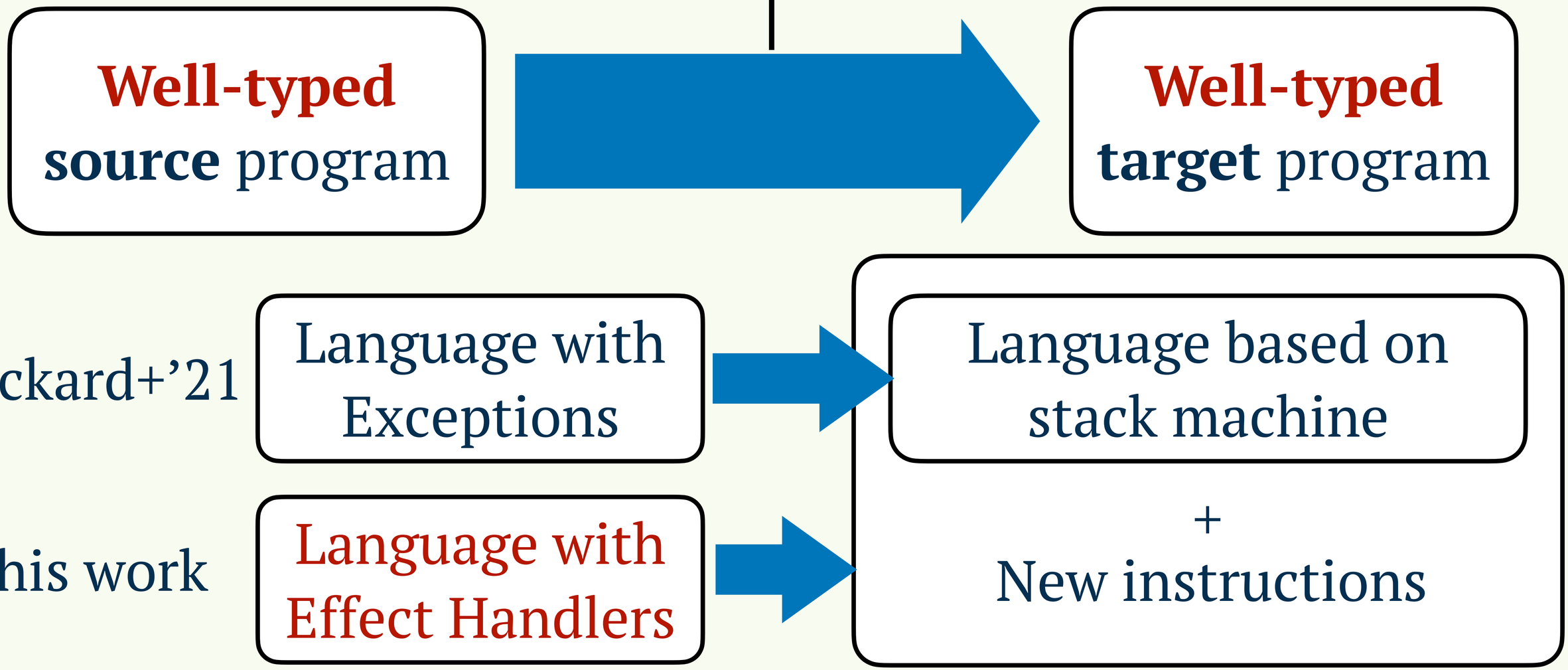
_ + _ , 1 hand

is bound to **k**

Target stack



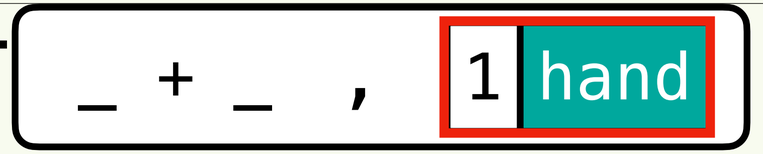
Background : Type-Preserving Compilation



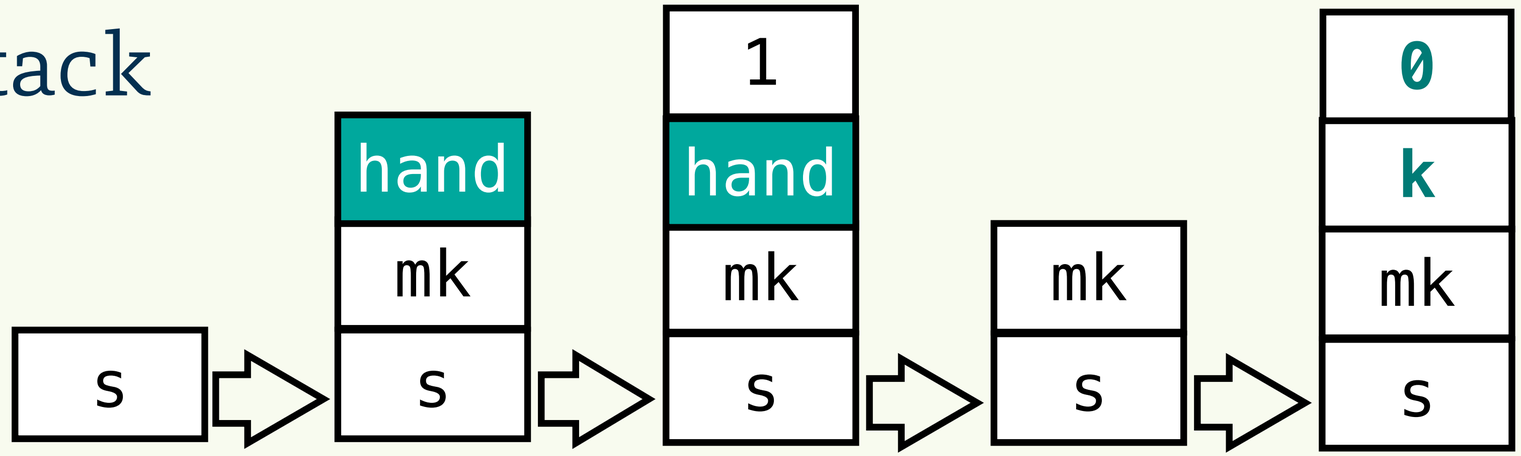
Effect Handlers in Target Language

Source term

```
handle _ + _
with {...; op _ k -> k 0 }
```



Target stack

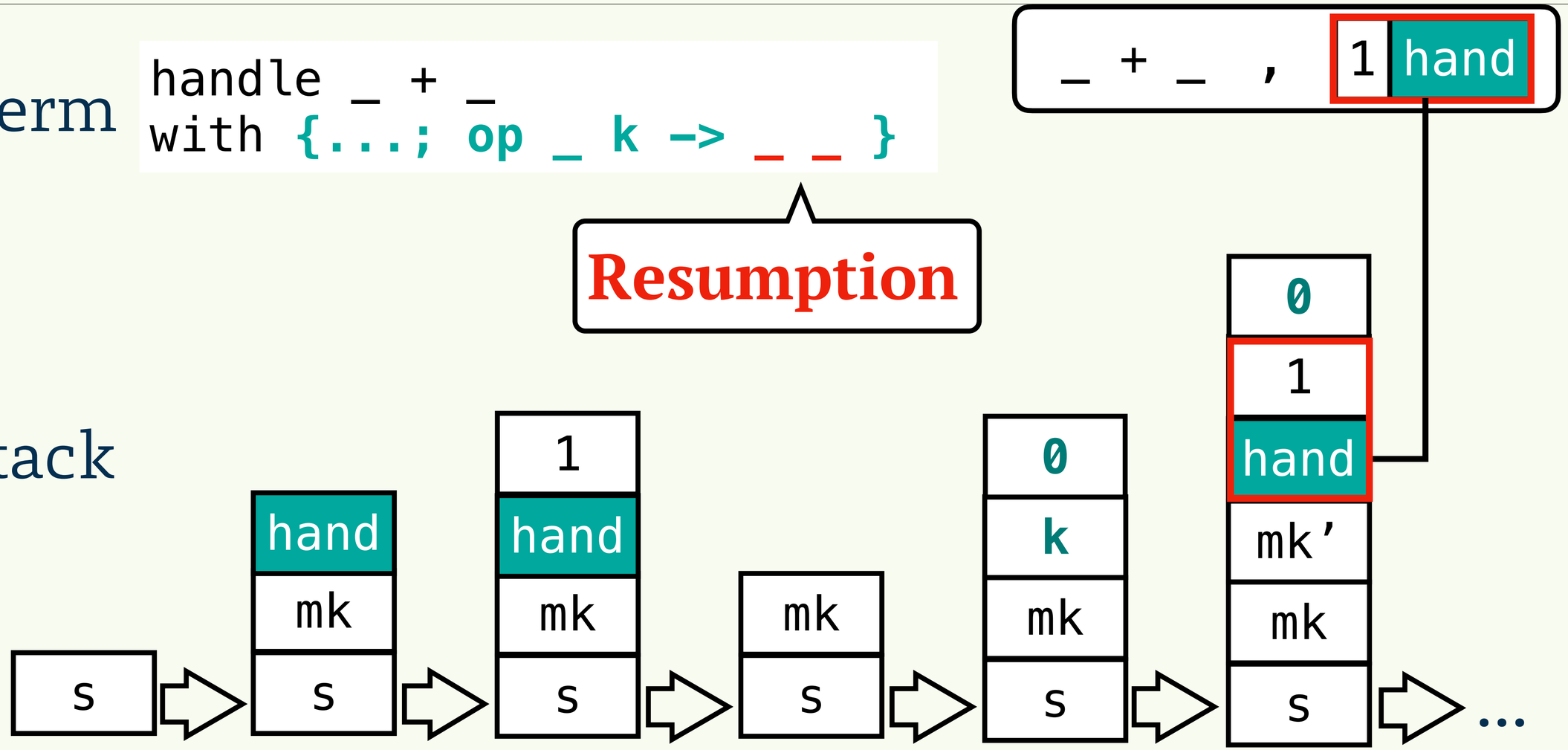


Effect Handlers in Target Language

Source term

```
handle _ + _
with {...; op _ k -> _ _ }
```

Resumption



Source Language : A Language with Effect Handlers

```
handle
  let x = do get () in
  return x
with {
  return x -> return x;
  get p k -> k 0
}
```

Operations cause effects

Handlers determine behavior of operations.

Handlers can use continuaions

Source Language : A Language with Effect Handlers

Execution example

```
handle
  let x = do get () in
  return x
with {
  return x -> return x;
  get p k -> k 0
}
```

Captured Continuation

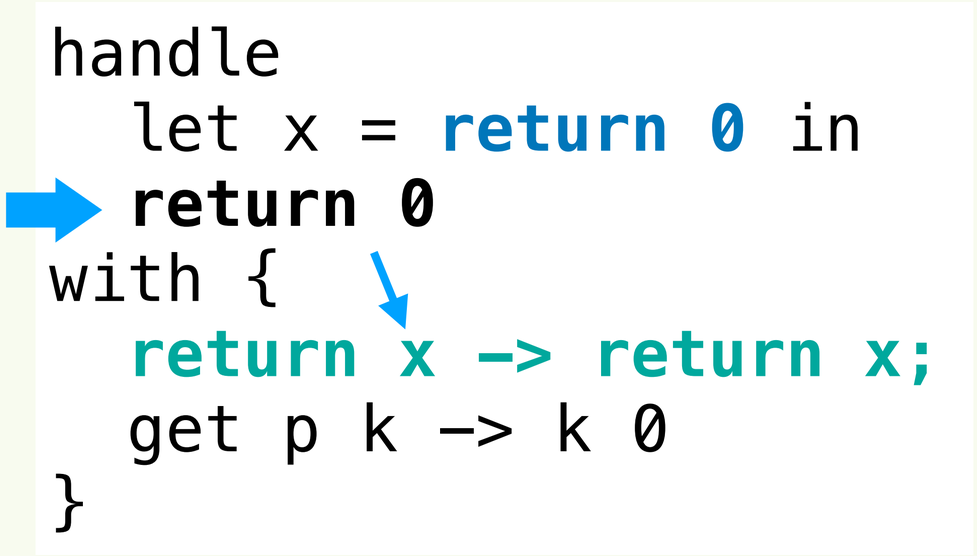
```
handle
  let x = return ? in
  return x
with ...
```

Resuming handled computation with `0`

return x -> return (x + 1)

Source Language : A Language with Effect Handlers

Execution example



=> 0

Source Language / Intrisically typed AST

Data types only representing well-typed terms in L_s

```
data Val (Γ : Ctx) : VTy → Set
data Cmp (Γ : Ctx) : CTy → Set
```

Γ ⊢ V : A

Γ ⊢ M : C

```
data Cmp Γ where
  App : Val Γ (A ⇒ C) → Val Γ A → Cmp Γ C
  ...
```

$$\frac{\Gamma \vdash v : A \Rightarrow C \qquad \Gamma \vdash w : A}{\Gamma \vdash v\ w : C}\ [\text{T–APP}]$$

Target Language : Intrisically typed AST

```
data Code (Γ : Ctx) : StackTy → StackTy → Set
```

```
Γ ⊢ M : (A , ε)
```

```
data Code (Γ : Ctx) : StackTy → StackTy → Set

OperationCodes : VTy → Eff → Eff → Ctx → StackTy → StackTy → Set
HandlerCode : Ctx → CTy → CTy → Set

PureCodeCont : Ctx → StackTy → CTy → Set
PureCodeCont Γ S₁ C =
  ∀{Γ₁ S₂ S₃} → Code Γ (S₁ ++ HandTy Γ₁ S₂ S₃ C :: S₂) S₃

OperationCodes B E₁ E₂ Γ SS S₃ = All (λ {(op A' B') → Code ((B' ⇒ (B , E₂)) :: A' :: Γ) SS S₃ }) E₁

HandlerCode Γ (A , E₁) (B , E₂) =
  (∀{Γ₁ Γ'₁ S₁ S₂ S₃ A'} →
    -- return clause
    Code (A :: Γ) (ContTy Γ'₁ (ValTy B :: S₁) (A' , E₂) :: (S₁ ++ HandTy Γ₁ S₂ S₃ (A' , E₂) :: S₂)) S₃ ×
    -- operation clauses
    OperationCodes B E₁ E₂ Γ (ContTy Γ'₁ (ValTy B :: S₁) (A' , E₂) :: (S₁ ++ HandTy Γ₁ S₂ S₃ (A' , E₂) :: S₂)) S₃
  )
```

```
{-# TERMINATING #-}
exec : Code Γ S S' → Stack S → RuntimeEnv Γ → Stack S'
exec (PUSH v c) s = exec c $ (val (pval v)) :: s
exec (ABS c' c) s env = exec c (val (clos c' env) :: s) env
exec (LOOKUP x c) s env = exec c ((val $ lookup env x) :: s) env
exec (APP c) (val v :: val (clos c' env') :: s) env = exec c' (cont c env :: s) (v :: env')
exec (APP c) (v :: val (fc-resump (c' , s' , env₂) (h , envh)) :: s) env =
  exec c' (v :: (s' ++s (hand h envh :: cont c env :: s))) env₂
exec (CALLOP l c) (val v :: s) env with split s
... | (s1 , (hand h env') , s2) with h
... | (_ , ops) = exec (lookup ops l) s2 (fc-resump (c , s1 , env) (h , env') :: v :: env')
exec (BIND c k) (val v :: s) env = exec c (cont k env :: s) (v :: env)
exec RET (val v :: cont c env :: s) _ = exec c (val v :: s) env
exec (MARK h mk c) s env = exec c (hand h env :: cont mk env :: s) env
exec (UNMARK) (val x :: (hand h env') :: s) env with h
... | (ret , ops) = exec ret s (x :: env')
exec (UNMARK) (val x :: init-hand :: s) env = val x :: s
exec (INITHAND c) s env = exec c (init-hand :: s) env
```

```
data Code Γ S S' → Stack S → RuntimeEnv Γ → Stack S'
  PUSH : PV
  ABS :
    -- function body
    (∀{S₁ S₂ S₃ Γ₁ Γ'₁ A'} → Code (A :: Γ) (ContTy Γ₁ (ValTy B :: S₁) (A' , E₁) :: (S₁ ++ HandTy Γ'₁ S₂ S₃ (A' , E₁) :: S₂)) S₃)  →
    Code Γ (ValTy (A ⇒ (B , E₁)) :: S) S' → Code Γ S S'
  LOOKUP : A ∈ Γ → Code Γ (ValTy A :: S) S' → Code Γ S S'
  APP : PureCodeCont Γ (ValTy B :: S₁) (A' , E) →
        Code Γ (ValTy A :: ValTy (A ⇒ (B , E)) :: S₁ ++ HandTy Γ₁ S₂ S₃ (A' , E) :: S₂) S₃
  CALLOP : (op A B) ∈ E
           → PureCodeCont Γ (ValTy B :: S₁) (A' , E)
           → Code Γ (ValTy A :: S₁ ++ HandTy Γ₁ S S' (A' , E) :: S) S'
```

```
data SValTy : Set
StackTy : Set

data SValTy where
  ValTy  : VTy → SValTy
  ContTy : Ctx → StackTy → CTy → SValTy
  HandTy : Ctx → StackTy → StackTy → CTy → SValTy

StackTy = List SValTy

variable
  T : SValTy
  S S' S₁ S₂ S₃ : StackTy


-- immediate values
data PVal : VTy → Set where
  unit : PVal Unit

_++s_ : Stack S → Stack S' → Stack (S ++ S')
[] ++s s = s
(x :: xs) ++s s = x :: (xs ++s s)

infixr 20 _++s_

split : Stack (S₁ ++ HandTy Γ₁ S S' (A , E) :: S) → Stack S₁ × Stack
split {S₁ = []} (H :: S) = ([] , H , S)
split {S₁ = _ :: _} (V :: S) with split S
```

```
data EnvVal : VTy → Set
RuntimeEnv : Ctx → Set

data StackVal : SValTy → Set

Stack : StackTy → Set
Stack S = All (λ T → StackVal T) S

data EnvVal where
  pval : PVal A → EnvVal A
  clos : (∀{Γ₁ Γ'₁ S₁ S₂ S₃ A'} → Code (A :: Γ) (ContTy Γ₁ (ValTy B :: S₁) (A' , E) :: (S₁ ++ HandTy Γ'₁ S₂ S₃ (A' , E) :: S₂)) S₃)
         → RuntimeEnv Γ → EnvVal (A ⇒ (B , E))
  fc-resump :
    PureCodeCont Γ (ValTy A :: S) (A' , E) × Stack S × RuntimeEnv Γ → -- continuation body and its stores
    HandlerCode Γ₁ (A' , E) (B , E') × RuntimeEnv Γ₁ → -- handler and its environment
    EnvVal (A ⇒ (B , E'))
```

Intrinsically Typed Compiler for Effect Handlers

Compiler for top-level program

```
compile : Cmp Γ (A , []) → Code Γ S (A :: S)
```

CPS Compiler for effectful computation

```
compileC :
  Cmp Γ (A , E) →
  (∀{S S'} → Code ... ) →
  Code Γ (S₁ ++ HandTy Γ₁ S S' (A' , E) :: S) S'
```

Continuation in target language

Intrinsically Typed Compiler for Effect Handlers

Compiler for top-level program

```
compile : Cmp Γ (A , []) → Code Γ S (A :: S)
```

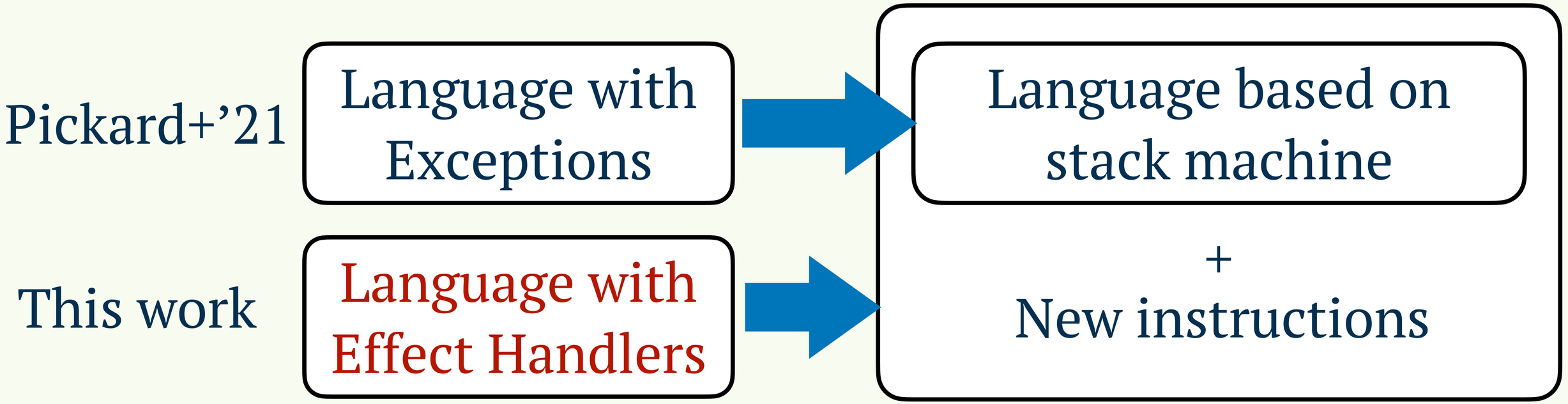
CPS Compiler for effectful computation

```
compileC :
  Cmp Γ (A , E) →
  (∀{S S'} → Code ... ) →
  Code Γ (S₁ ++ HandTy Γ₁ S S' (A' , E) :: S) S'
```

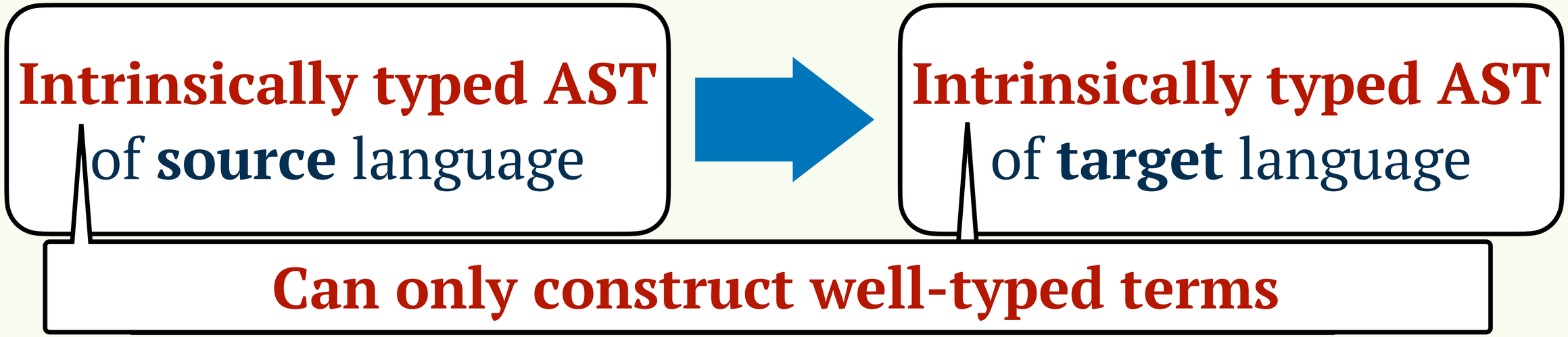
Continuation in target language

Type checked definition of function `compile`
= **Proof** of type preservation of compiler

Approach

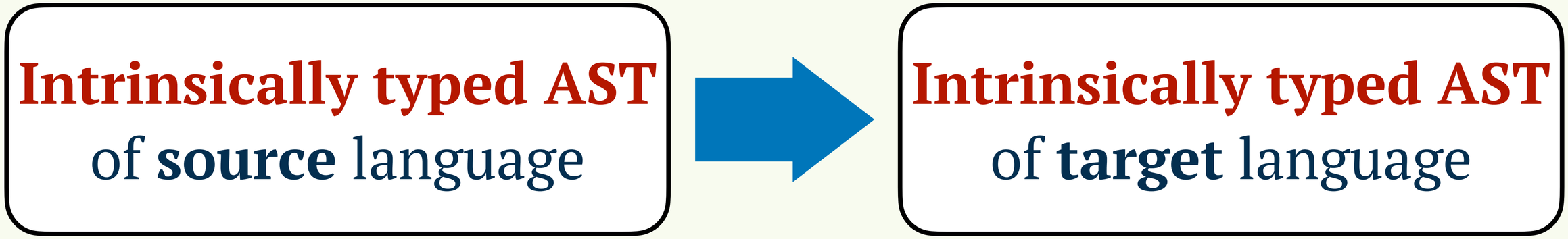

Pickard+'21
Language with Exceptions
This work
Language with Effect Handlers
Language based on stack machine
+
New instructions

Background : Intrinsically Typed Compiler



Type checked definition = Proof of type preservation

Background : Intrinsically Typed AST



Simple AST

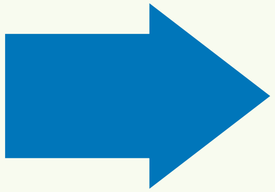
```
data Exp :       Set where
  num  : ℕ → Exp
  bool : 𝔹 → Exp
  add  : Exp     → Exp     → Exp
```

Agda

```
add (num 1) (bool true) : Exp
```

Background : Intrinsically Typed AST

Intrinsically typed AST of **source** language



Intrinsically typed AST of **target** language

Intrinsically-typed AST

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

Agda

```
add (num 1) (bool true) : Exp Nat
```

Goal

Develop type-preserving compilers for languages with first-class continuations

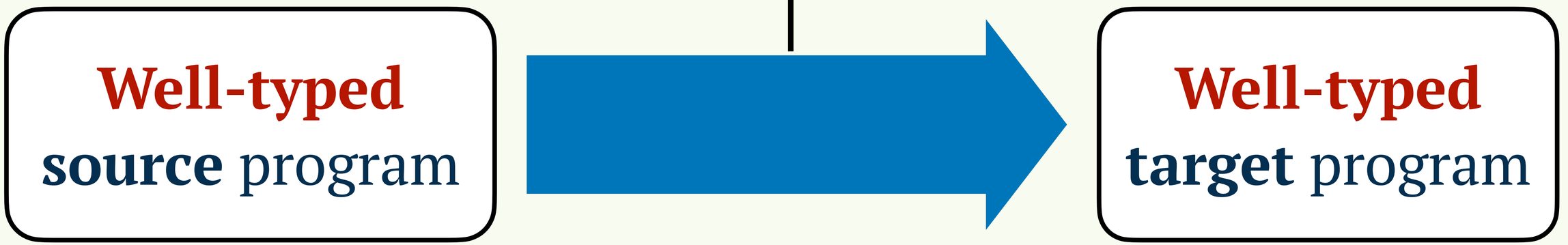
Previous work[Morrisett+'99, Chlipala'07, Pickard+'21] considers

Functions | Conditionals & loops | Exceptions

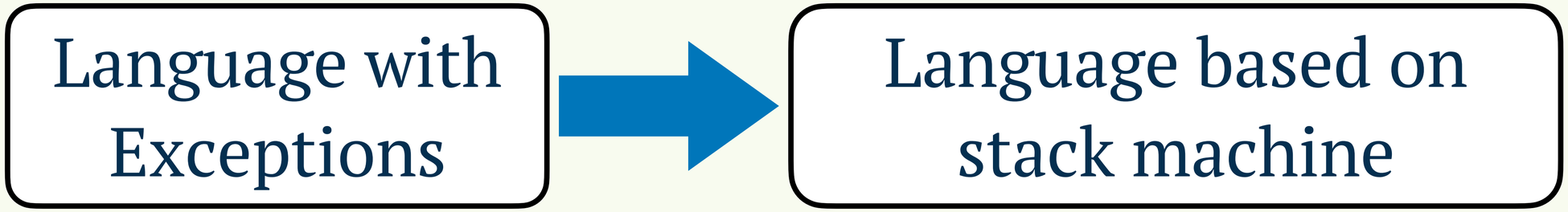
This work considers Effect handlers

Making an intrinsically typed compiler for

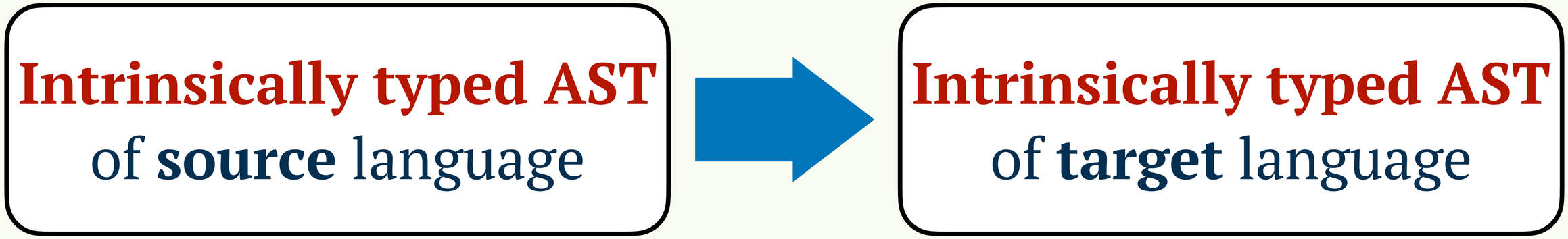
Background : Type-Preserving Compilation



Pickard+'21



Background : Intrinsically Typed AST



Simple AST

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

Agda

```
add (num 1) (bool true) : Exp
```

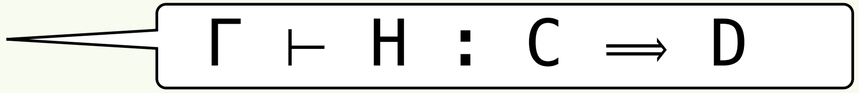
Source Language : Intrinsically typed AST

Data types only representing well-typed terms

```
data Val (Γ : Ctx) : VTy → Set
```


```
data Cmp (Γ : Ctx) : CTy → Set
```


```
data Hdl (Γ : Ctx) : HTy → Set
```




```
data VTy : Set  -- Value types
CTy : Set       -- Computation types
data HTy : Set  -- Handler types

data Sig : Set  -- Effect signatures
Eff : Set       -- Effects

data Sig where
  op : VTy → VTy → Sig

Eff = List Sig
```

```
data VTy where
  Unit  : VTy
  _⇒_   : VTy → CTy →
          VTy
  Hand  : HTy → VTy

CTy = VTy × Eff

data HTy where
  _⟹_ : CTy → CTy → HTy

Ctx = List VTy

variable
  A B A' B' : VTy
  E E' E₁ E₂ : Eff
  C D : CTy
  H : HTy
  Γ Γ' Γ₁ Γ₂ : Ctx
```

```
data Val (Γ : Ctx) : VTy → Set
data Cmp (Γ : Ctx) : CTy → Set
data Hdl (Γ : Ctx) : HTy → Set
OperationClauses : Ctx → Eff → CTy → Set

data Val Γ where
  Unit : Val Γ Unit
  Var : A ∈ Γ → Val Γ A
  Lam : Cmp (A :: Γ) C → Val Γ (A ⇒ C)

data Cmp Γ where
  Return : Val Γ A → Cmp Γ (A , E)
  Do : (op A B) ∈ E → Val Γ A → Cmp Γ (B , E)
  Handle_With_  : Cmp Γ C → Hdl Γ (C ⟹ D) → Cmp Γ D
  App : Val Γ (A ⇒ C) → Val Γ A → Cmp Γ C
  Let_In_ : Cmp Γ (A , E) → (Cmp (A :: Γ) (B , E))
                          → Cmp Γ (B , E)
```

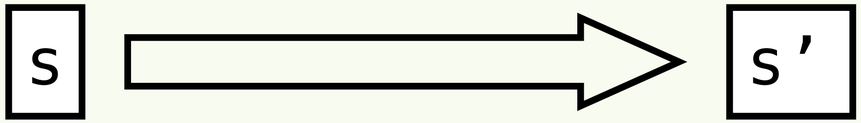
```
data Hdl Γ where
  ǩx_|ǩx,r_ :
    Cmp (A :: Γ) C → -- return clause
    OperationClauses Γ E C → -- operation clauses
    Hdl Γ ((A , E) ⟹ C)

OperationClauses Γ E₁ D =
  All (λ { (op A' B') → Cmp ((B' ⇒ D) :: A' :: Γ) D }) E₁
```

Incorporating Stack Polymorphism

Typing a term in target language (Monomorphic)

$\Gamma \vdash$ `code : S ⟹ S'`



Target Language : Intrinsically typed AST

```
data Code (Γ : Ctx) : StackTy → StackTy → Set
```

Γ ⊢ code : S ⟹ S’

```
data Code Γ where
  PUSH : PVal A → Code Γ (ValTy A :: S) S' → Code Γ S S'
  ABS :
    -- function body
    (∀{S₁ S₂ S₃ Γ₁ Γ'₁ A'} → Code (A :: Γ) (ContTy Γ₁ (ValTy B :: S₁) (A' , E₁) :: (S₁ ++ HandTy Γ'₁ S₂ S₃ (A' , E₁) :: S₂)) S₃)  →
    Code Γ (ValTy (A ⇒ (B , E₁)) :: S) S' → Code Γ S S'
  LOOKUP : A ∈ Γ → Code Γ (ValTy A :: S) S' → Code Γ S S'
  APP : PureCodeCont Γ (ValTy B :: S₁) (A' , E) →
        Code Γ (ValTy A :: ValTy (A ⇒ (B , E)) :: S₁ ++ HandTy Γ₁ S₂ S₃ (A' , E) :: S₂) S₃
  CALLOP : (op A B) ∈ E
           → PureCodeCont Γ (ValTy B :: S₁) (A' , E)
           → Code Γ (ValTy A :: S₁ ++ HandTy Γ₁ S S' (A' , E) :: S) S'
  BIND : Code (A :: Γ) (ContTy Γ (ValTy B :: S) C :: (S ++ HandTy Γ₂ S₂ S₃ C :: S₂)) S₃ -- let body
       → PureCodeCont Γ (ValTy B :: S) C → Code Γ (ValTy A :: (S ++ HandTy Γ₂ S₂ S₃ C :: S₂)) S₃
  RET : Code Γ (ValTy A :: ContTy Γ₁ (ValTy A :: S) C :: (S ++ HandTy Γ₂ S₂ S₃ C :: S₂)) S₃
  MARK :
    HandlerCode Γ (A , E₁) (B , E₂) → -- handler
    PureCodeCont Γ (ValTy B :: S₁) (B' , E₂) → -- meta-continuation
    -- handled computation
    Code Γ (
      HandTy Γ (ContTy Γ (ValTy B :: S₁) (B' , E₂) :: S₁ ++ HandTy Γ₁ S₂ S₃ (B' , E₂) :: S₂) S₃ (A , E₁) ::
               ContTy Γ (ValTy B :: S₁) (B' , E₂) :: S₁ ++ HandTy Γ₁ S₂ S₃ (B' , E₂) :: S₂
    ) S₃ →
    Code Γ (S₁ ++ HandTy Γ₁ S₂ S₃ (B' , E₂) :: S₂) S₃
  UNMARK : Code Γ (ValTy A :: HandTy Γ₁ S S' (A , E₁) :: S) S'
  INITHAND : Code Γ (HandTy Γ S (ValTy A :: S) (A , []) :: S) (ValTy A :: S) → Code Γ S (ValTy A :: S)
```

```
{-# TERMINATING #-}
exec : Code Γ S S' → Stack S → RuntimeEnv Γ → Stack S'
exec (PUSH v c) s = exec c $ (val (pval v)) :: s
exec (ABS c' c) s env = exec c (val (clos c' env) :: s) env
exec (LOOKUP x c) s env = exec c ((val $ lookup env x) :: s) env
exec (APP c) (val v :: val (clos c' env') :: s) env = exec c' (cont c env :: s) (v :: env')
exec (APP c) (v :: val (fc-resump (c' , s' , env₂) (h , envh)) :: s) env =
  exec c' (v :: (s' ++s (hand h envh :: cont c env :: s))) env₂
exec (CALLOP l c) (val v :: s) env with split s
... | (s1 , (hand h env') , s2) with h
... | (_ , ops) = exec (lookup ops l) s2 (fc-resump (c , s1 , env) (h , env') :: v :: env')
exec (BIND c k) (val v :: s) env = exec c (cont k env :: s) (v :: env)
exec RET (val v :: cont c env :: s) _ = exec c (val v :: s) env
exec (MARK h mk c) s env = exec c (hand h env :: cont mk env :: s) env
exec (UNMARK) (val x :: (hand h env') :: s) env with h
... | (ret , ops) = exec ret s (x :: env')
exec (UNMARK) (val x :: init-hand :: s) env = val x :: s
exec (INITHAND c) s env = exec c (init-hand :: s) env
```