

# GHCによる Haskellプログラムの動かかし方

Haskell Day 2021

# おことわり

- 時間の都合上, Haskell の基本的な知識と, 計算機への理解があることを前提にしています
- 説明の都合上, 最適化や内部表現の違いにより説明と実態に齟齬が生じる場合があります
- GHC 9.0 / Haskell 2010 を前提とします

# 今日話すこと

- GHC が採用している抽象機械 STG について
- Haskell プログラムと STG の対応について

# 今日話さないこと

- Haskell プログラムに対する型推論や型検査について
- GHC が採用する GC・ランタイムシステムについて

# このプログラムはどう動く？

```
main :: IO ()
main =
    let ones = 1:ones
        go z0 xs0 = case xs0 of
            _ | z0 >= 100 -> z0
            []             -> z0
            x:xs           -> go (x + z0) xs
    in print (go 0 ones :: Int)
```

# このプログラムはどう動く？

```
main :: IO ()
main =
    let ones = 1:ones
        go z0 xs0 = case xs0 of
            _ | z0 >= 100 -> z0
            []             -> z0
            x:xs           -> go (x + z0) xs
    in print (go 0 ones :: Int)
```

1 が無限に続くリスト i.e. [1, 1, 1, ...]

# このプログラムはどう動く？

```
main :: IO ()
main =
    let ones = 1:ones
        go z0 xs0 = case xs0 of
            _ | z0 >= 100 -> z0
            []             -> z0
            x:xs           -> go (x + z0) xs
    in print (go 0 ones :: Int)
```

go 0 ones を表すサックを作成

# このプログラムはどう動く？

```
main :: IO ()
main =
  let ones = 1:ones
      go z0 xs0 = case xs0 of
                    | z0 >= 100 -> z0
                    -> z0
                    -> go (x + z0) xs
  in print (go 0 ones :: Int)
```

渡されたサックを評価

x:xs

# このプログラムはどう動く？

```
main :: IO ()
main =
  let ones = 1
      go z0 xs0 = case xs0 of
                    _ | z0 >= 100 -> z0
                    []           -> z0
                    x:xs         -> go (x + z0) xs
  in print (go 0 ones :: Int)
```

z0 を評価し, 100 以上か判定



# このプログラムはどう動く？

```
main :: IO ()
main =
  let ones = 1
  go z0 xs0 = case xs0 of
    - | z0 >= 100 -> z0
    [] -> go (x + z0) xs
  in print (go 0 ones :: Int)
```

z0 を評価し, 100 以上か判定

xs0 を評価し, パターンマッチ

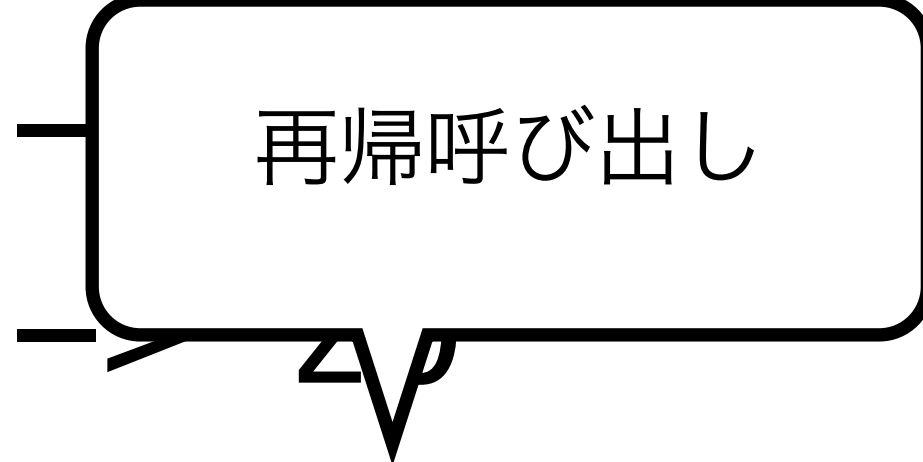
# このプログラムはどう動く？

```
main :: IO ()
main =
    let ones = 1:ones
        go z0 xs0 = case xs0 of
            _ | z0 >= 100 -> z0
            []             -> z0
            x:xs           -> go (x + z0) xs
    in print (go 0 ones :: Int)
```

x + z0 を表すサックを作成

# このプログラムはどう動く？

```
main :: IO ()
main =
  let ones = 1:ones
      go z0 xs0 = case xs0 of
                    _ | z0 >= 100 -> []
                    x:xs         -> go (x + z0) xs
  in print (go 0 ones :: Int)
```



# このプログラムはどう動く？

```
main :: IO ()
main =
    let ones = 1:ones
        go z0 xs0 = case xs0 of
            _ | z0 >= 100 -> z0
            []             -> z0
            x:xs           -> go (x + z0) xs
    in print (go 0 ones :: Int)
```

引数を返却

# このプログラムはどう動く？

```
main :: IO ()
main =
  let ones = 1:ones
      go z0 xs0 = case xs0 of
                    | z0 >= 100 -> z0
                    -> z0
                    -> go (x + z0) xs
  in print (go 0 ones :: Int)
```

整数を表示

# GHC のパイプライン

## Haskell プログラム

```
main :: IO ()  
main = putStrLn "Hello, World!"
```

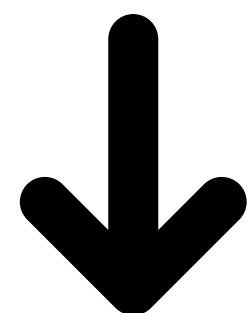
構文解析

型検査 + 型推論

脱糖衣

最適化

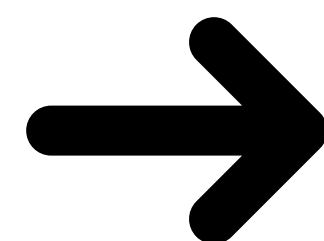
単純化



## Core (System F $\omega$ )

単純化された Haskell プログラム

変換

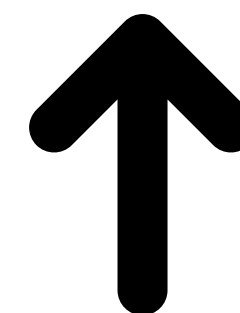


## 実行ファイル

exec

リンク

メモリ管理部分追記



機械語生成

## STG

抽象機械による実行モデル

# GHC のパイプライン

## Haskell プログラム

```
main :: IO ()  
main = putStrLn "Hello, World!"
```

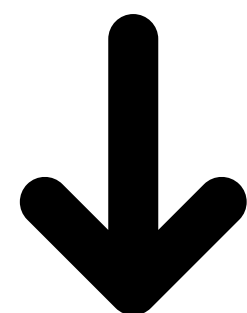
構文解析

型検査 + 型推論

脱糖衣

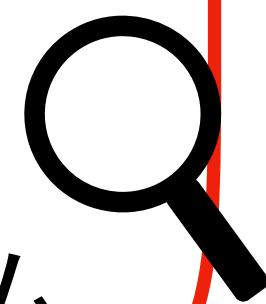
最適化

単純化

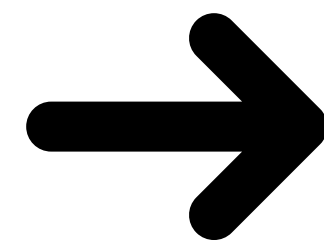


## Core (System F $\omega$ )

単純化された Haskell プログラム



変換

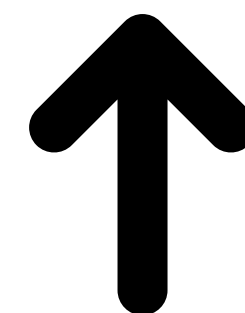


## 実行ファイル

exec

リンク

メモリ管理部分追記



機械語生成

## STG

抽象機械による実行モデル

# GHC のパイプライン

## Haskell プログラム

```
main :: IO ()
main = putStrLn "Hello, World!"
```

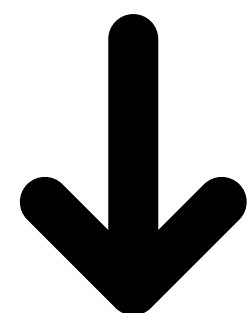
構文解析

型検査 + 型推論

脱糖衣

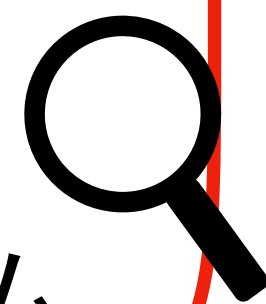
最適化

単純化

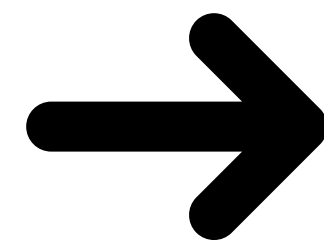


## Core (System F $\omega$ )

単純化された Haskell プログラム



変換

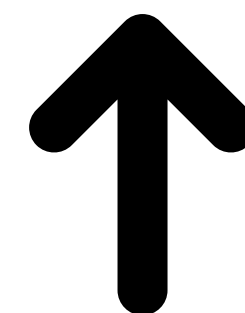


## 実行ファイル

exec

リンク

メモリ管理部分追記



機械語生成

## STG

抽象機械による実行モデル



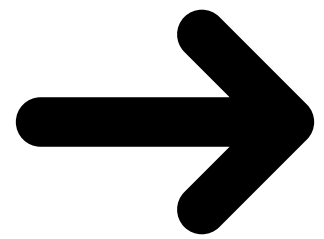
# Haskell の機能

if 式	if ... then ... else ...	
case 式	case ... of { ... }	ガード節 複雑なパターンマッチ
let 式	let { ... } in ...	ガード節 パターンマッチによる束縛 引数を伴った束縛
do 式	do { ... }	パターンマッチによる束縛
ラムダ抽象	\ ... -> ...	ガード節 パターンマッチによる束縛
関数適用	f x1 x2 ...	中置記法 レコード構文
etc.	...	型クラス リスト内包表記



# Haskell → Core

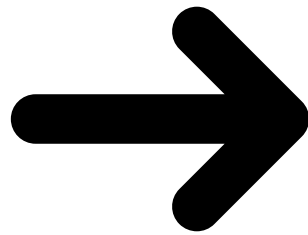
if 式	if ... then ... else ...	
case 式	case ... of { ... }	ガード節 複雑なパターンマッチ
let 式	let { ... } in ...	ガード節 パターンマッチによる束縛 引数を伴った束縛
do 式	do { ... }	パターンマッチによる束縛
ラムダ抽象	\ ... -> ...	ガード節 パターンマッチによる束縛
関数適用	f x1 x2 ...	中置記法 レコード構文
etc.	...	型クラス リスト内包表記



case 式	case ... of { ... }	パターンマッチはネスト禁止 ガード節禁止
let 式	let { ... } in ...	単なる束縛のみ可能
ラムダ抽象	\ ... -> ...	パターンマッチ禁止 ガード節禁止
関数適用	f x1 x2 ...	中置記法禁止 レコード構文禁止
cast 式	... `cast` ...	newtype / GADTs に対する 操作で使用 (今回は触れない)

# Haskell → Core

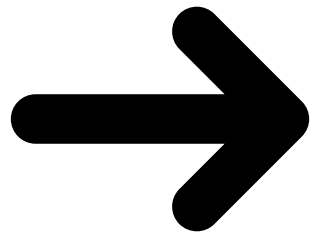
if 式	if ... then ... else ...	
True / False で分岐する case 式に		ガード節 パターンマッチ
let 式	let { ... } in ...	ガード節 パターンマッチによる束縛 引数を伴った束縛
do 式	do { ... }	パターンマッチによる束縛
ラムダ抽象	\ ... -> ...	ガード節 パターンマッチによる束縛
関数適用	f x1 x2 ...	中置記法 レコード構文
etc.	...	型クラス リスト内包表記



case 式	case ... of { ... }	パターンマッチはネスト禁止 ガード節禁止
let 式	let { ... } in ...	単なる束縛のみ可能
ラムダ抽象	\ ... -> ...	パターンマッチ禁止 ガード節禁止
関数適用	f x1 x2 ...	中置記法禁止 レコード構文禁止
cast 式	... `cast` ...	newtype / GADTs に対する 操作で使用 (今回は触れない)

# Haskell → Core

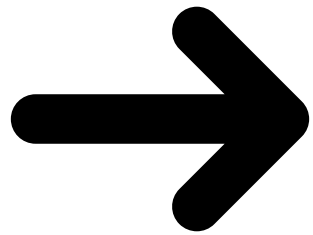
if 式	if ... then ... else ...	
case 式	case ... of { ... }	ガード節 複雑なパターンマッチ
<div>評価順序を考慮して、ネスト・ガード節を紐解き、ネストしない形のパターンしか使わない必ず評価が必要になる case 式のネストに</div>		
ラムダ抽象	\ ... -> ...	ガード節 パターンマッチによる束縛
関数適用	f x1 x2 ...	中置記法 レコード構文
etc.	...	型クラス リスト内包表記



case 式	case ... of { ... }	パターンマッチはネスト禁止 ガード節禁止
let 式	let { ... } in ...	単なる束縛のみ可能
ラムダ抽象	\ ... -> ...	パターンマッチ禁止 ガード節禁止
関数適用	f x1 x2 ...	中置記法禁止 レコード構文禁止
cast 式	... `cast` ...	newtype / GADTs に対する 操作で使用 (今回は触れない)

# Haskell → Core

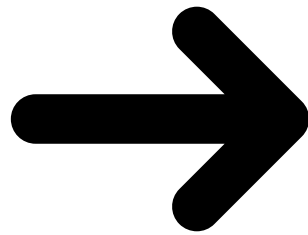
if 式	if ... then ... else ...	
case 式	case ... of { ... }	ガード節 複雑なパターンマッチ
let 式	let { ... } in ...	ガード節 パターンマッチによる束縛 引数を伴った束縛
単純な束縛を行う let 式と、その後パターンマッチを行う case 式の組み合わせに		
関数適用	f x1 x2 ...	中置記法 レコード構文
etc.	...	型クラス リスト内包表記



case 式	case ... of { ... }	パターンマッチはネスト禁止 ガード節禁止
let 式	let { ... } in ...	単なる束縛のみ可能
ラムダ抽象	\ ... -> ...	パターンマッチ禁止 ガード節禁止
関数適用	f x1 x2 ...	中置記法禁止 レコード構文禁止
cast 式	... `cast` ...	newtype / GADTs に対する 操作で使用 (今回は触れない)

# Haskell → Core

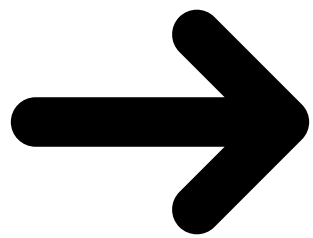
if 式	if ... then ... else ...	
case 式	case ... of { ... }	ガード節 複雑なパターンマッチ
let 式	let { ... } in ...	ガード節 パターンマッチによる束縛 引数を伴った束縛
do 式	do { ... }	パターンマッチによる束縛
モナド	モナド	パターンマッチによる束縛
モナドのメソッドを使い、パターンマッチも case 式を使って明示的に書くように		
etc.	...	型クラス リスト内包表記



case 式	case ... of { ... }	パターンマッチはネスト禁止 ガード節禁止
let 式	let { ... } in ...	単なる束縛のみ可能
ラムダ抽象	\ ... -> ...	パターンマッチ禁止 ガード節禁止
関数適用	f x1 x2 ...	中置記法禁止 レコード構文禁止
cast 式	... `cast` ...	newtype / GADTs に対する 操作で使用 (今回は触れない)

# Haskell → Core

if 式	if ... then ... else ...	
case 式	case ... of { ... }	ガード節 複雑なパターンマッチ
let 式	let { ... } in ...	ガード節 パターンマッチによる束縛 引数を伴った束縛
do 式	do { ... }	パターンマッチによる束縛
ラムダ抽象	\ ... -> ...	ガード節 パターンマッチによる束縛
中置記法		
let 式の時と同様に、パターンマッチは case 式で行うように		

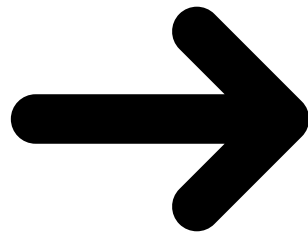


case 式	case ... of { ... }	パターンマッチはネスト禁止 ガード節禁止
let 式	let { ... } in ...	単なる束縛のみ可能
ラムダ抽象	\ ... -> ...	パターンマッチ禁止 ガード節禁止
関数適用	f x1 x2 ...	中置記法禁止 レコード構文禁止
cast 式	... `cast` ...	newtype / GADTs に対する 操作で使用 (今回は触れない)



# Haskell → Core

if 式	if ... then ... else ...	
case 式	case ... of { ... }	ガード節 複雑なパターンマッチ
let 式	let { ... } in ...	ガード節 パターンマッチによる束縛 引数を伴った束縛
中置記法・レコード構文は同等のただの関数適用に変換		
関数適用	f x1 x2 ...	中置記法 レコード構文
etc.	...	型クラス リスト内包表記

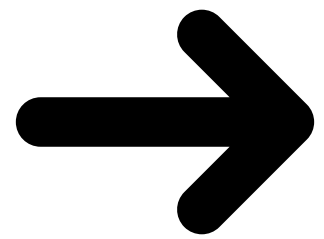


case 式	case ... of { ... }	パターンマッチはネスト禁止 ガード節禁止
let 式	let { ... } in ...	単なる束縛のみ可能
ラムダ抽象	\ ... -> ...	パターンマッチ禁止 ガード節禁止
関数適用	f x1 x2 ...	中置記法禁止 レコード構文禁止
cast 式	... `cast` ...	newtype / GADTs に対する 操作で使用 (今回は触れない)



# Haskell → Core

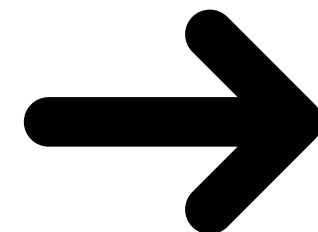
if 式	if ... then ... else ...	
case 式	case ... of { ... }	ガード節 複雑なパターンマッチ
let 式	let { ... } in ...	ガード節 パターンマッチによる束縛 引数を伴った束縛
do 式	do { ... }	パターンマッチによる束縛
ラムダ抽象	\ ... -> ...	ガード節 パターンマッチによる束縛
関数適用	f x1 x2 ...	中置記法 レコード構文
etc.	...	型クラス リスト内包表記



case 式	case ... of { ... }	パターンマッチはネスト禁止 ガード節禁止
let 式	let { ... } in ...	単なる束縛のみ可能
ラムダ抽象	\ ... -> ...	パターンマッチ禁止 ガード節禁止
関数適用	f x1 x2 ...	中置記法禁止 レコード構文禁止
cast 式	... `cast` ...	newtype / GADTs に対する 操作で使用 (今回は触れない)

# プログラムの整理

```
main :: IO ()
main =
  let ones = 1:ones
      go z0 xs0 = case xs0 of
        _ | z0 >= 100 -> z0
        []             -> z0
        x:xs           -> go (x + z0) xs
  in print (go 0 ones :: Int)
```



```
main =
  let ones = (:) 1 ones in -- メモリに保存
  let go = \z0 xs0 -> case (>=) z0 100 of -- 評価
    True -> z0
    False -> case xs0 of -- 評価
      [] ->
        z0
      (:) x xs ->
        let a1 = (+) x z0 in -- サンク作成してメモリに保存
        go a1 xs
  in
  let go_0_ones = go 0 ones in -- サンク作成してメモリに保存
  print go_0_ones
```

# GHC のパイプライン

## Haskell プログラム

```
main :: IO ()
main = putStrLn "Hello, World!"
```

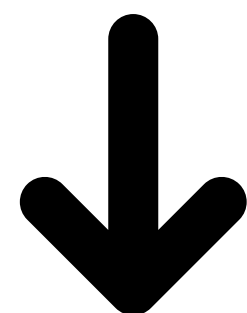
構文解析

型検査 + 型推論

脱糖衣

最適化

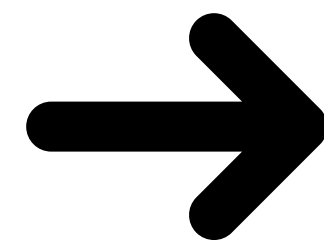
単純化



## Core (System F $\omega$ )

単純化された Haskell プログラム

変換

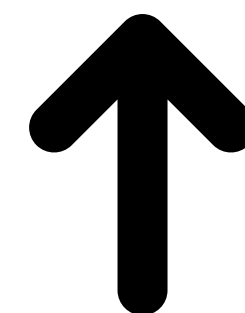


## 実行ファイル

exec

リンク

メモリ管理部分追記



機械語生成

## STG

抽象機械による実行モデル

# GHC のパイプライン

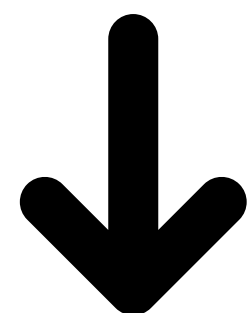
## Haskell プログラム

```
main :: IO ()  
main = putStrLn "Hello, World!"
```

構文解析

脱糖衣

単純化



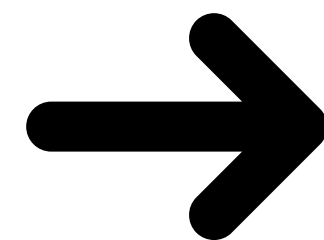
型検査 + 型推論

最適化

## Core (System F $\omega$ )

単純化された Haskell プログラム

変換

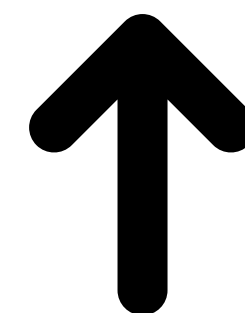


## 実行ファイル

exec

リンク

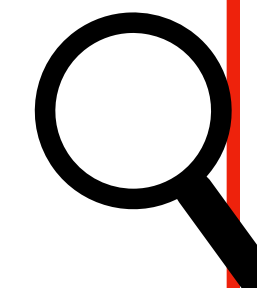
メモリ管理部分追記



機械語生成

## STG

抽象機械による実行モデル



# 実行モデルの基本

スタックとヒープで状態管理し， エントリーポイント呼び出しで遷移する典型的なマシン上で動く

case 式		式を評価しコンストラクタタグにより分岐
let 式	join point	エントリーポイント作成
		必要ならサンクを作成し， ヒープに保存
ラムダ抽象		エントリーポイント作成 & 必要ならクロージャ生成
関数適用	コンストラクタへの適用	ヒープオブジェクト生成
		適用先の関数を評価し， 引数が足りてないならクロージャ生成 引数が足りてるならジャンプ

# 実行モデルの基本

スタックとヒープで状態管理し， エントリーポイント呼び出しで遷移する典型的なマシン上で動く

case 式		式を評価しコンストラクタタグにより分岐
let 式	join point	エントリーポイント作成
		必要ならサ 関数呼び出しにスタックは消費しない！ ・他の言語には見られない特色 ・遅延評価だからこそできること
ラムダ抽象		エントリーポイント
関数適用	コンストラクタへの適用	ヒープオブジェクト生成
		引数が足りてるならジャンプ 適用先の関数を評価し， 引数が足りてないならクロージャ生成

# 実行モデルの基本

スタックとヒープで状態管理し， エントリーポイント呼び出しで遷移する典型的なマシン上で動く

case 式		式を評価しコンストラクタタグにより分岐
let 式	join point	エントリーポイント作成
		フレームを作成し， ヒープに保存
ラムダ抽象		エントリーポイント作成 & 必要ならクロージャ生成
関数適用	コンストラクタへの適用	ヒープオブジェクト生成
		適用先の関数を評価し， 引数が足りてないならクロージャ生成 引数が足りてるならジャンプ

# 実行モデルの基本

スタックとヒープで状態管理し， エントリーポイント呼び出しで遷移する典型的なマシン上で動く

case 式		式を評価しコンストラクタタグにより分岐
let 式	join point	エントリーポイント作成
		必要ならサンクを作成し， ヒープに保存
ラムダ抽象		エントリーポイント作成 & 必要ならクロージャ生成
関数適用	コンストラクタ	オブジェクト生成
		引数が足りてるならジャンプ 適用先の関数を評価し， 引数が足りてないならクロージャ生成

ヒープオブジェクト生成だけを行うような場合， サンクを作らず直接オブジェクト生成



# 実行モデルの基本

スタックとヒープで状態管理し， エントリーポイント呼び出しで遷移する典型的なマシン上で動く

case 式		式を評価しコンストラクタタグにより分岐
let 式	join point	エントリーポイント作成
		必要ならサンクを作成し， ヒープに保存
	エントリーポイント作成 & 必要ならクロージャ生成	
	ヒープオブジェクト生成	
関数適用		引数が足りてるならジャンプ
	適用先の関数を評価し，	引数が足りてないならクロージャ生成

クロージャ生成が不要なく， エントリーポイントに直接飛べば良い場合は， エントリーポイントだけ作成（今回は詳しく触れない）

# 実行モデルの基本

スタックとヒープで状態管理し， エントリーポイント呼び出しで遷移する典型的なマシン上で動く

case 式		式を評価しコンストラクタタグにより分岐
let 式	join point	エントリーポイント作成
		必要ならサンクを作成し， ヒープに保存
ラムダ抽象		エントリーポイント作成 & 必要ならクロージャ生成
関数適用	コンストラクタへの適用	ヒープオブジェクト生成
		適用先の関数を評価し， 引数が足りてないならクロージャ生成 引数が足りてるならジャンプ

# GHC が採用する抽象機械 STG

- 実行モデルに対応する型無しの抽象機械
- Shared Term Graph の頭文字を取っている (\*)

式  $e$  の種類

let 式	$\text{let } \{x_1 = obj_1; \dots; x_n = obj_n\} \text{ in } e$
case 式	$\text{case } e \text{ of } \{alt_1; \dots; alt_n\}$
関数適用	$f\ a_1 \ \dots \ a_n$
アトムック値	$a$

ヒープオブジェクト  $obj$  の種類

関数	$\text{FUN}(x_1 \ \dots \ x_n \rightarrow e)$
クロージャ	$\text{PAP}(f\ a_1 \ \dots \ a_n)$
データ値	$\text{CON}(C\ a_1 \ \dots \ a_n)$
サンク	$\text{THUNK}(e)$

アトムック値  $a$  の種類

変数	$x$
リテラル	3.14

分岐  $alt$  の種類

$C\ x_1 \ \dots \ x_n \rightarrow e$
$x \rightarrow e$

(\*) <https://gitlab.haskell.org/ghc/ghc/-/blob/ghc-9.0.1-release/compiler/GHC/Stg/Syntax.hs#L4>

# let 式の意味論

```
let ones = CON((:) 1 ones) in ...
```

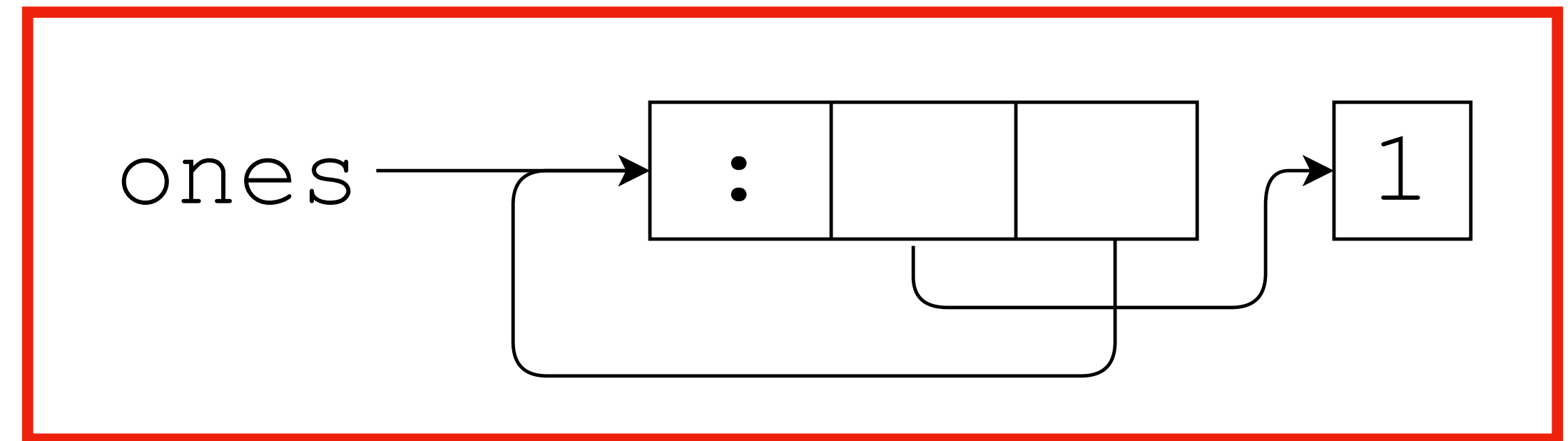
- ・ヒープオブジェクト作成
- ・ヒープオブジェクトへの参照の作成

# let 式の意味論

let ones = CON((:) 1 ones) in ...

- ・ヒープオブジェクト作成
- ・ヒープオブジェクトへの参照の作成

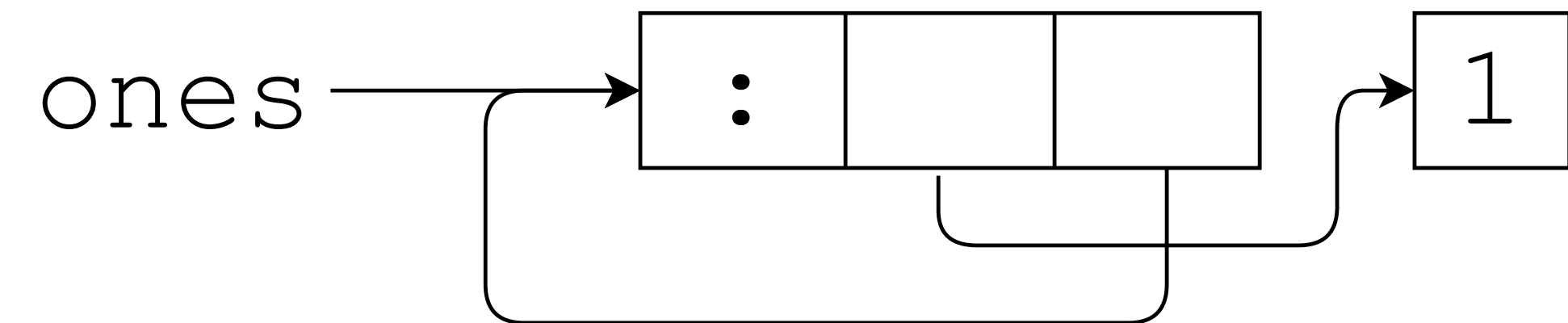
# let 式の意味論



`let ones = CON(:) 1 ones in ...`

- ・ヒープオブジェクト作成
- ・ヒープオブジェクトへの参照の作成

# let 式の意味論

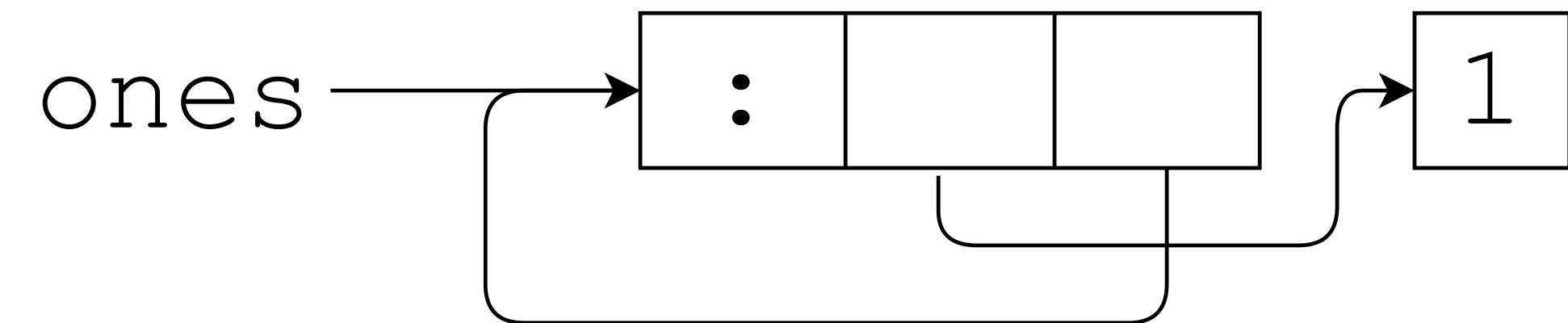


`let ones = CON( ( : ) 1 ones ) in ...`

`let go = FUN( z0 xs0 -> ... ) in ...`

- ・ヒープオブジェクト作成
- ・ヒープオブジェクトへの参照の作成

# let 式の意味論



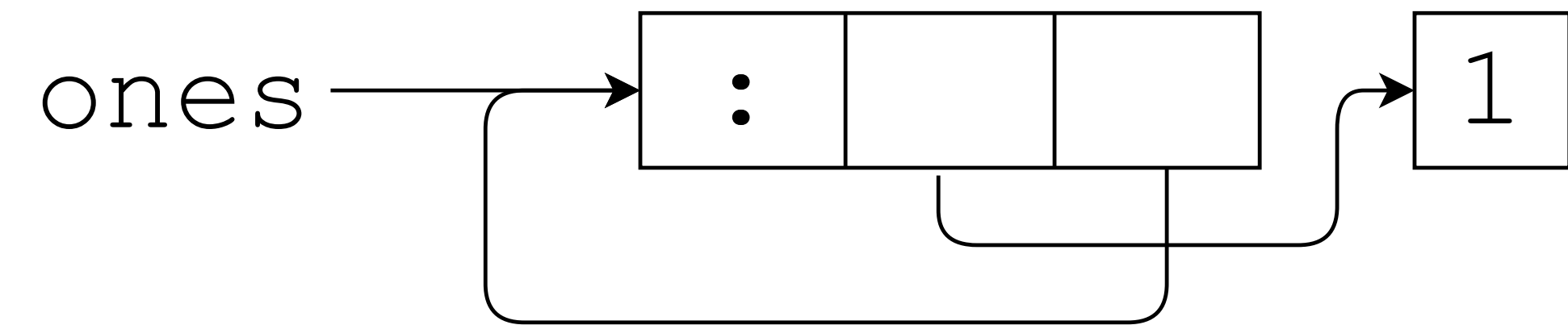
`let ones = CON( ( : ) 1 ones ) in ...`

`let go = FUN( z0 xs0 -> ... ) in ...`

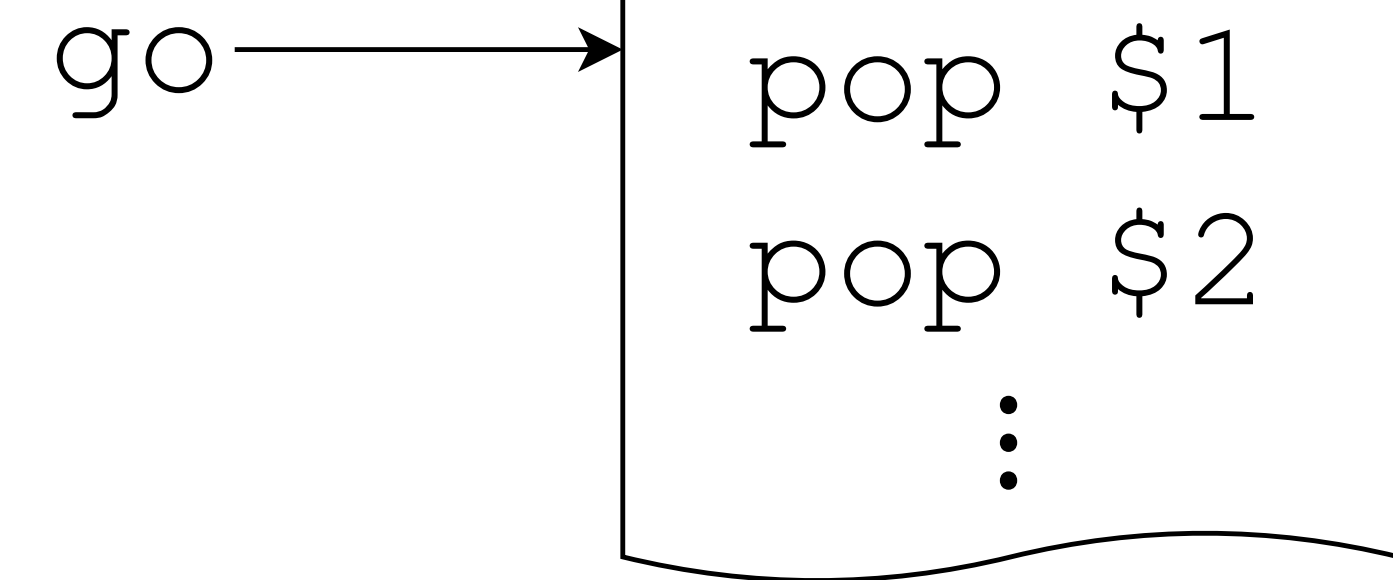
- ・ヒープオブジェクト作成
- ・ヒープオブジェクトへの参照の作成



# let 式の意味論



let ones = CON( ( : ) 1 ones ) in ...



let go = FUN( z0 xs0 -> ... ) in ...

- ・ヒープオブジェクト作成
- ・ヒープオブジェクトへの参照の作成

# let 式の意味論

- ・ヒープオブジェクト作成
- ・ヒープオブジェクトへの参照の作成

# let 式の意味論

```
let f' = FUN(n x -> (-) x n) in  
let f = PAP(f' 1) in ...
```

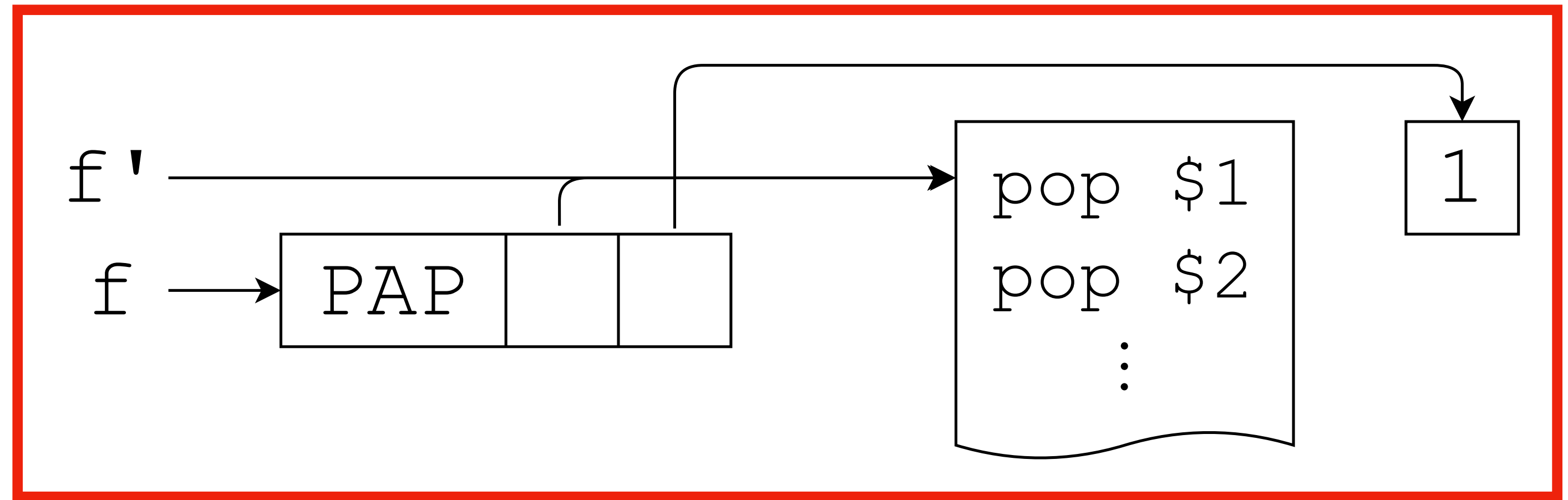
- ・ヒープオブジェクト作成
- ・ヒープオブジェクトへの参照の作成

# let 式の意味論

```
let f' = FUN(n x -> (-) x n) in  
let f = PAP(f' 1) in ...
```

- ・ヒープオブジェクト作成
- ・ヒープオブジェクトへの参照の作成

# let 式の意味論

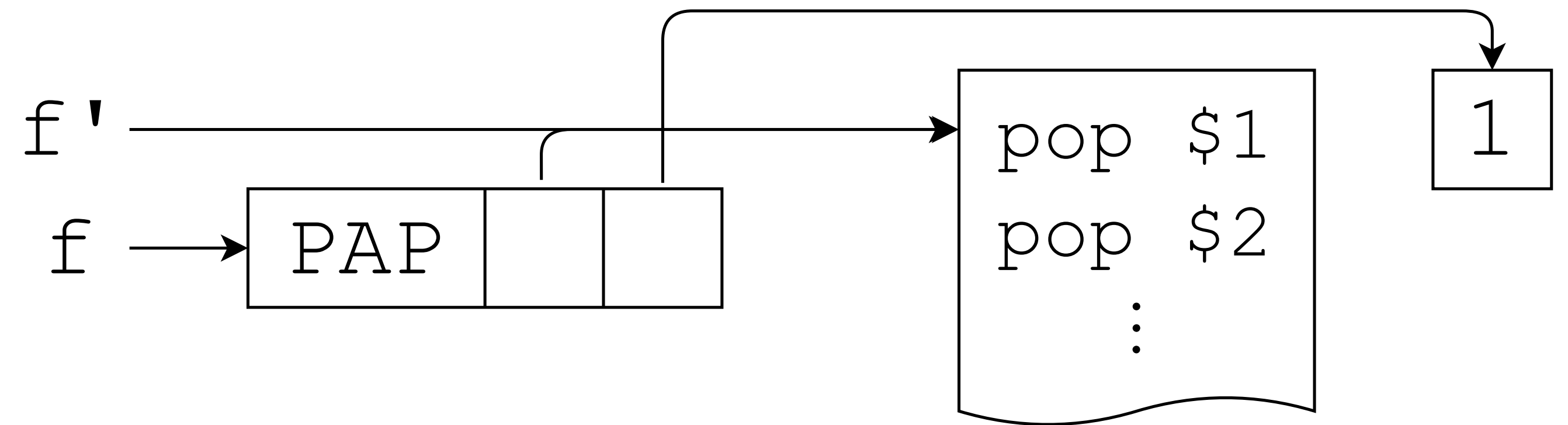


`let f' = FUN(n x -> (-) x n) in`

`let f = PAP(f' 1) in ...`

- ・ヒープオブジェクト作成
- ・ヒープオブジェクトへの参照の作成

# let 式の意味論

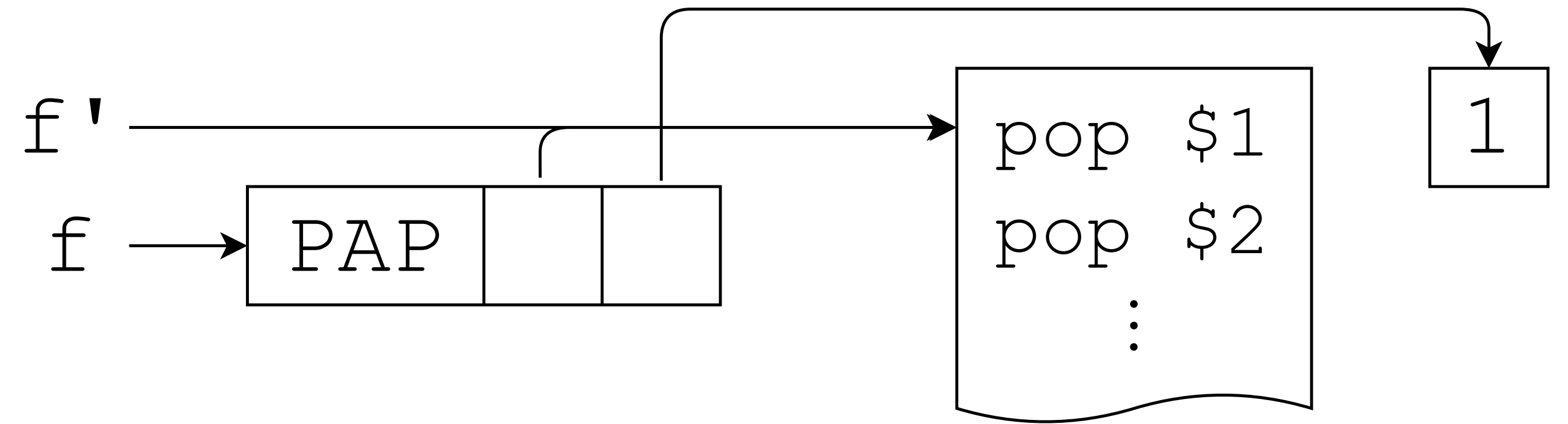


```
let f' = FUN(n x -> (-) x n) in  
let f = PAP(f' 1) in ...
```

```
let h = THUNK((+) 1 2) in ...
```

- ・ヒープオブジェクト作成
- ・ヒープオブジェクトへの参照の作成

# let 式の意味論

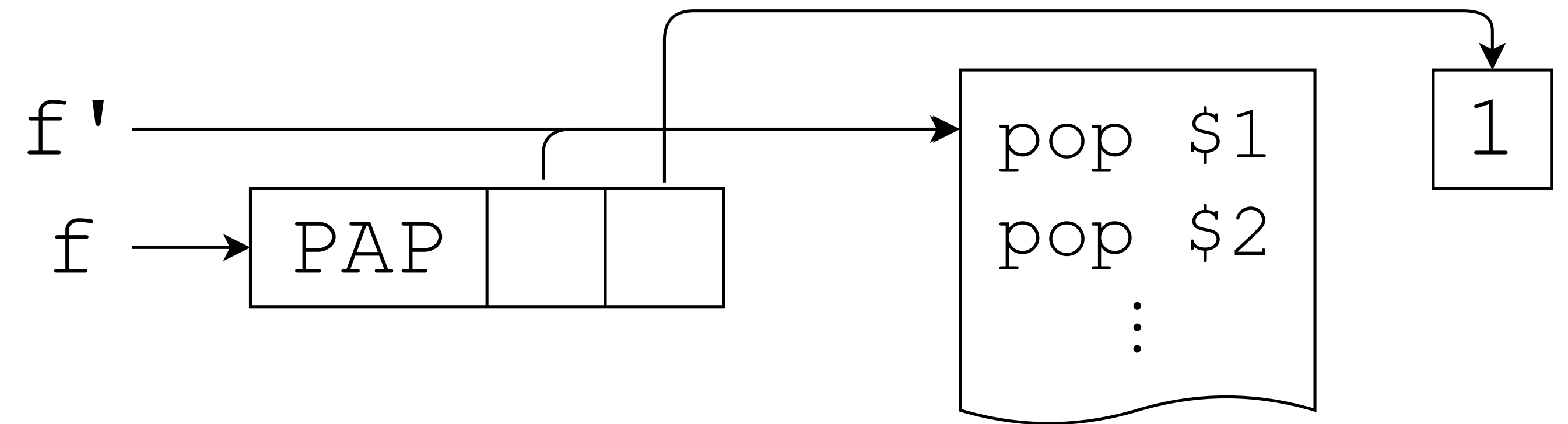


```
let f' = FUN(n x -> (-) x n) in  
let f = PAP(f' 1) in ...
```

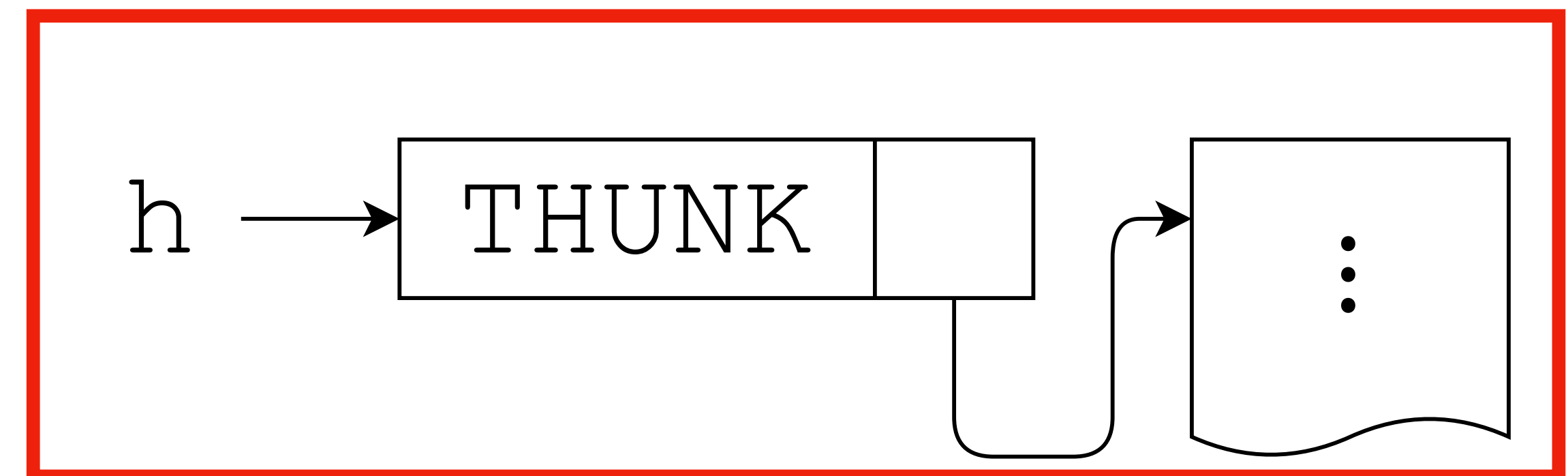
```
let h = THUNK((+) 1 2) in ...
```

- ・ヒープオブジェクト作成
- ・ヒープオブジェクトへの参照の作成

# let 式の意味論



```
let f' = FUN(n x -> (-) x n) in  
let f = PAP(f' 1) in ...
```



```
let h = THUNK((+) 1 2) in ...
```

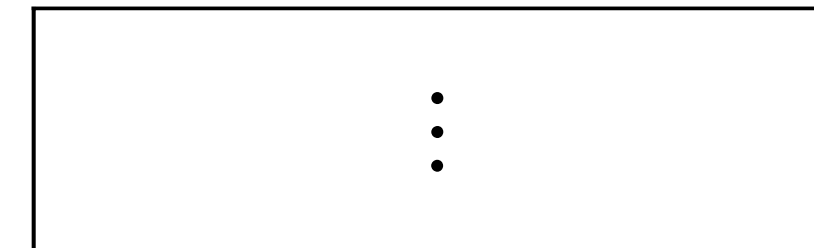
- ・ヒープオブジェクト作成
- ・ヒープオブジェクトへの参照の作成



# case 式の意味論

スタック

```
case (>=) z0 100 of {  
    True  -> z0;  
    False -> (*) z0 2;  
}
```

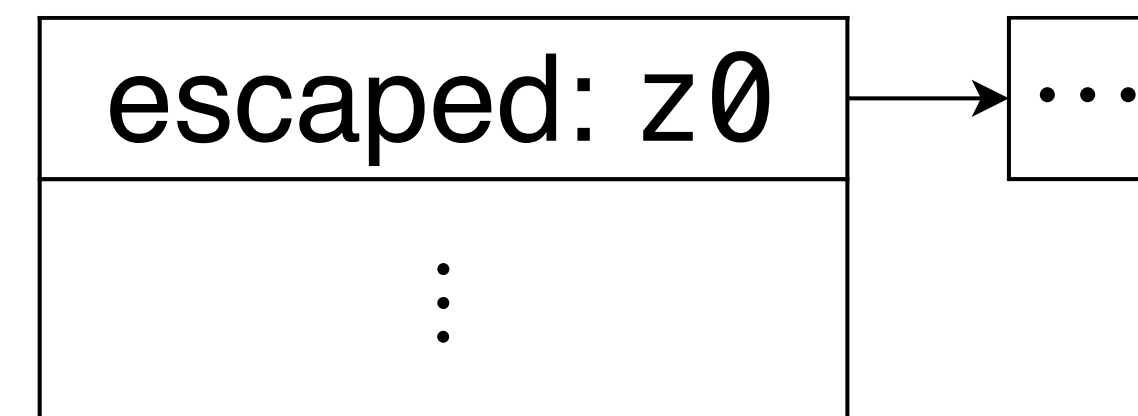


- ・スタックへの環境の退避と復元
- ・式の評価結果による分岐

# case 式の意味論

スタック

```
case (>=) z0 100 of {  
  True  -> z0;  
  False -> (*) z0 2;  
}
```

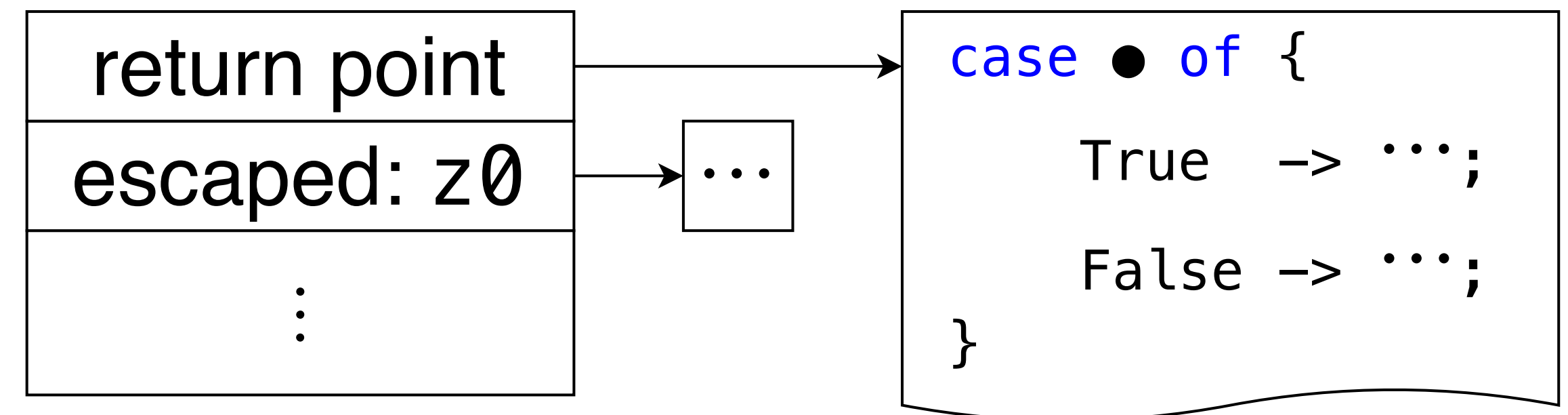


- スタックへの環境の退避と復元
- 式の評価結果による分岐

# case 式の意味論

```
case (>=) z0 100 of {  
  True  -> z0;  
  False -> (*) z0 2;  
}
```

スタック

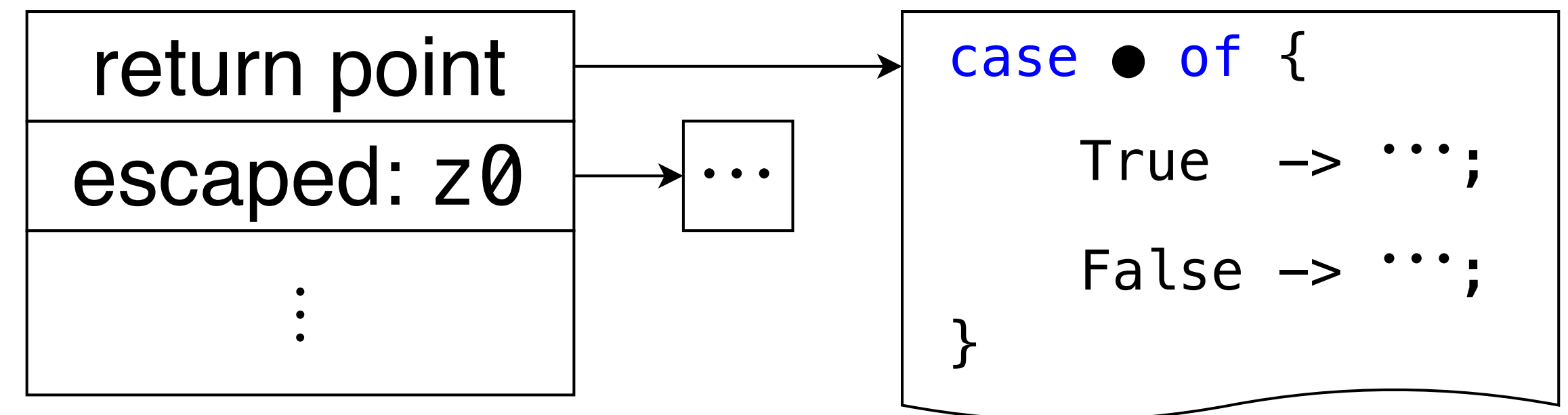


- スタックへの環境の退避と復元
- 式の評価結果による分岐

# case 式の意味論

```
case (>=) z0 100 of {  
  True  -> z0;  
  False -> (*) z0 2;  
}
```

スタック

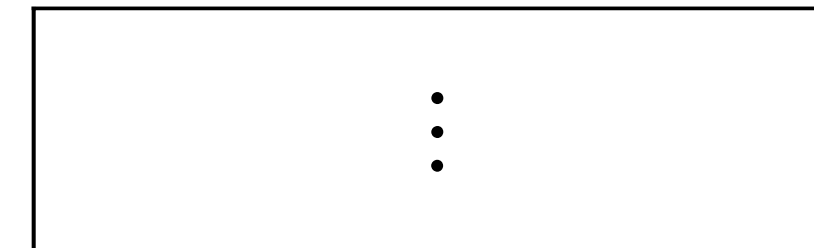


- スタックへの環境の退避と復元
- 式の評価結果による分岐

# case 式の意味論

スタック

```
case (>=) z0 100 of {  
  True  -> z0;  
  False -> (*) z0 2;  
}
```



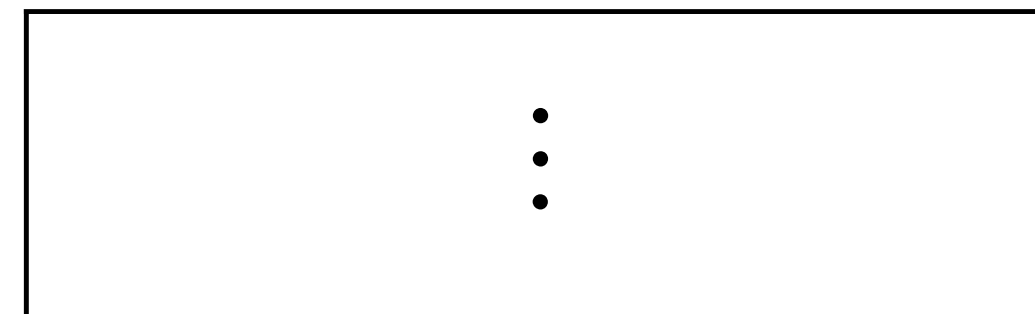
- スタックへの環境の退避と復元
- 式の評価結果による分岐

# サック評価の意味論

```
let h = THUNK((+) 1 2) in h
```

ヒープ

スタック



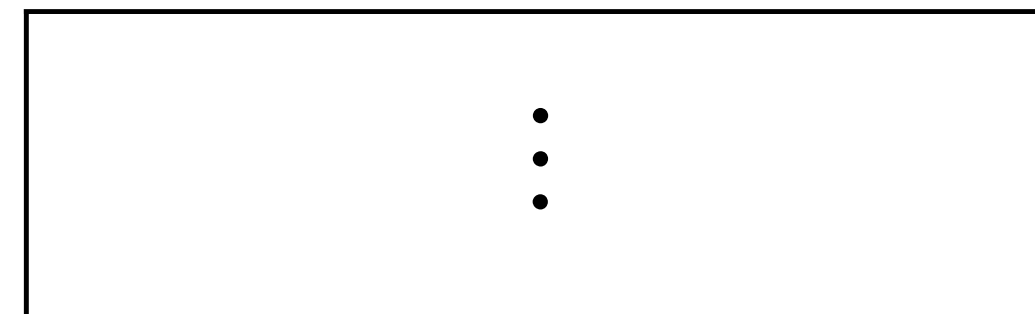
- ・サック式を評価
- ・結果への間接参照で更新

# サング評価の意味論

let h = THINK((+) 1 2) in h

ヒープ

スタック

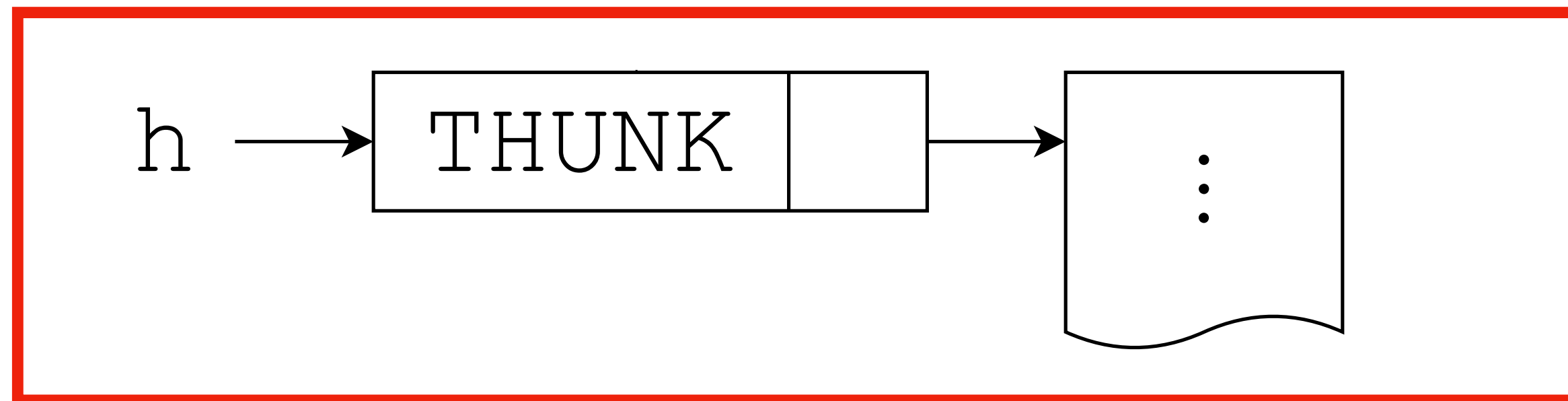


- ・サング式を評価
- ・結果への間接参照で更新

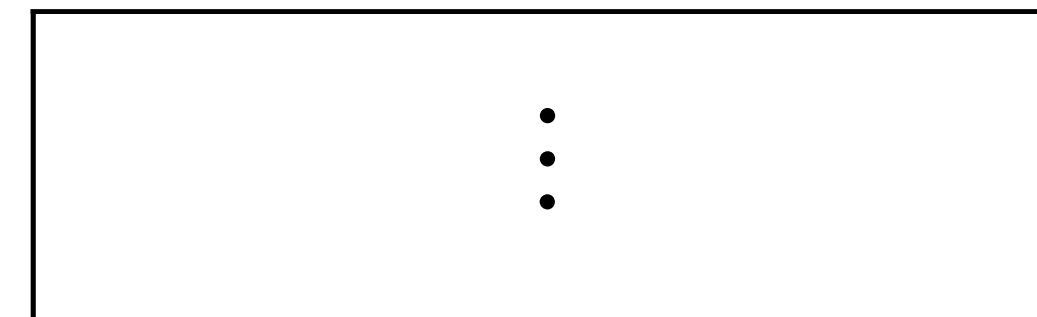
# サング評価の意味論

```
let h = THUNK( (+) 1 2) in h
```

ヒープ



スタック



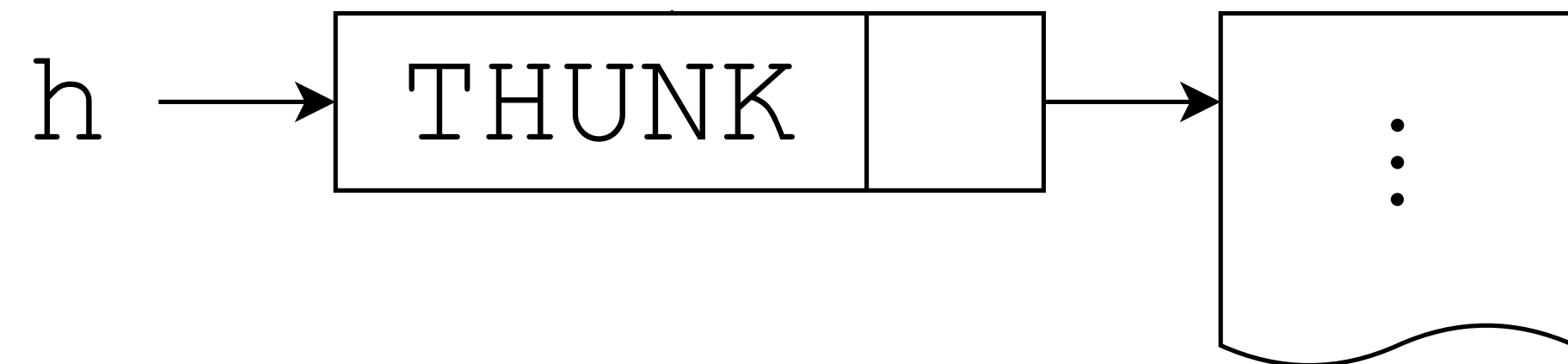
- ・サング式を評価
- ・結果への間接参照で更新



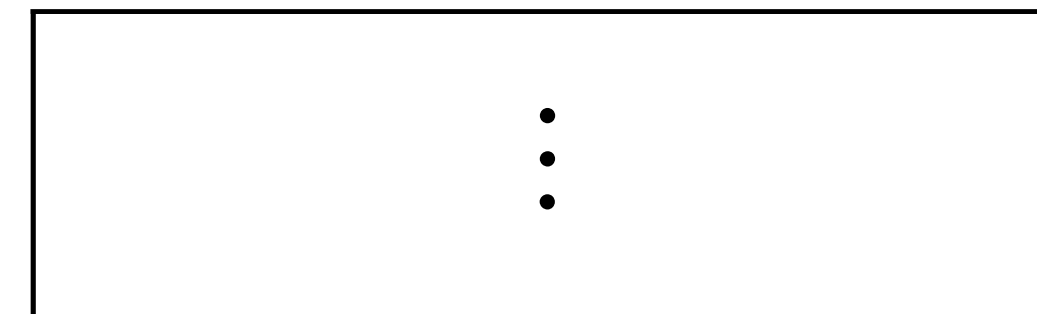
# サング評価の意味論

let h = THUNK( (+) 1 2) in h

ヒープ



スタック



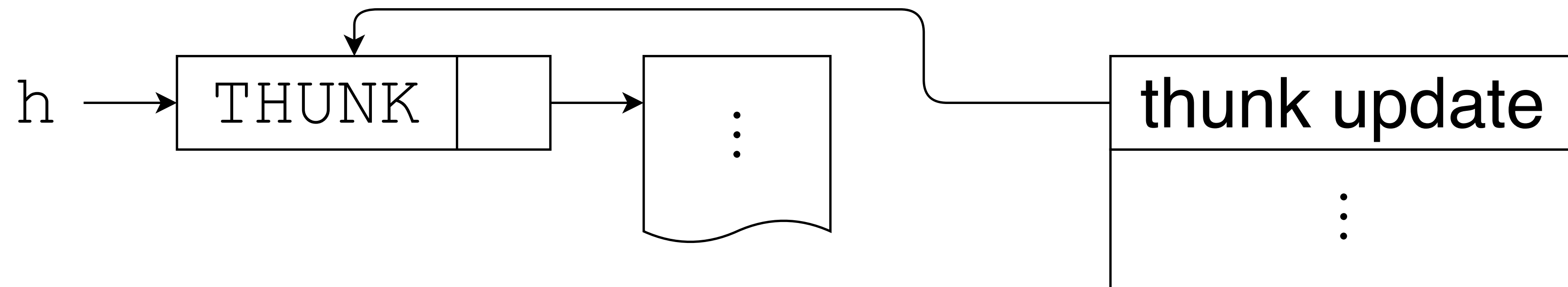
- ・サング式を評価
- ・結果への間接参照で更新

# サック評価の意味論

let h = THUNK( (+) 1 2 ) in h

ヒープ

スタック



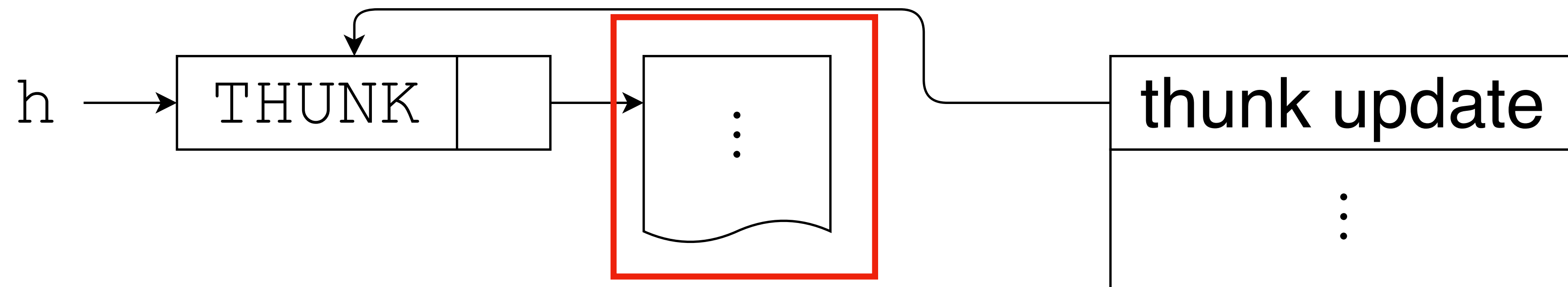
- ・サック式を評価
- ・結果への間接参照で更新

# サング評価の意味論

let h = THUNK((+) 1 2) in h

ヒープ

スタック

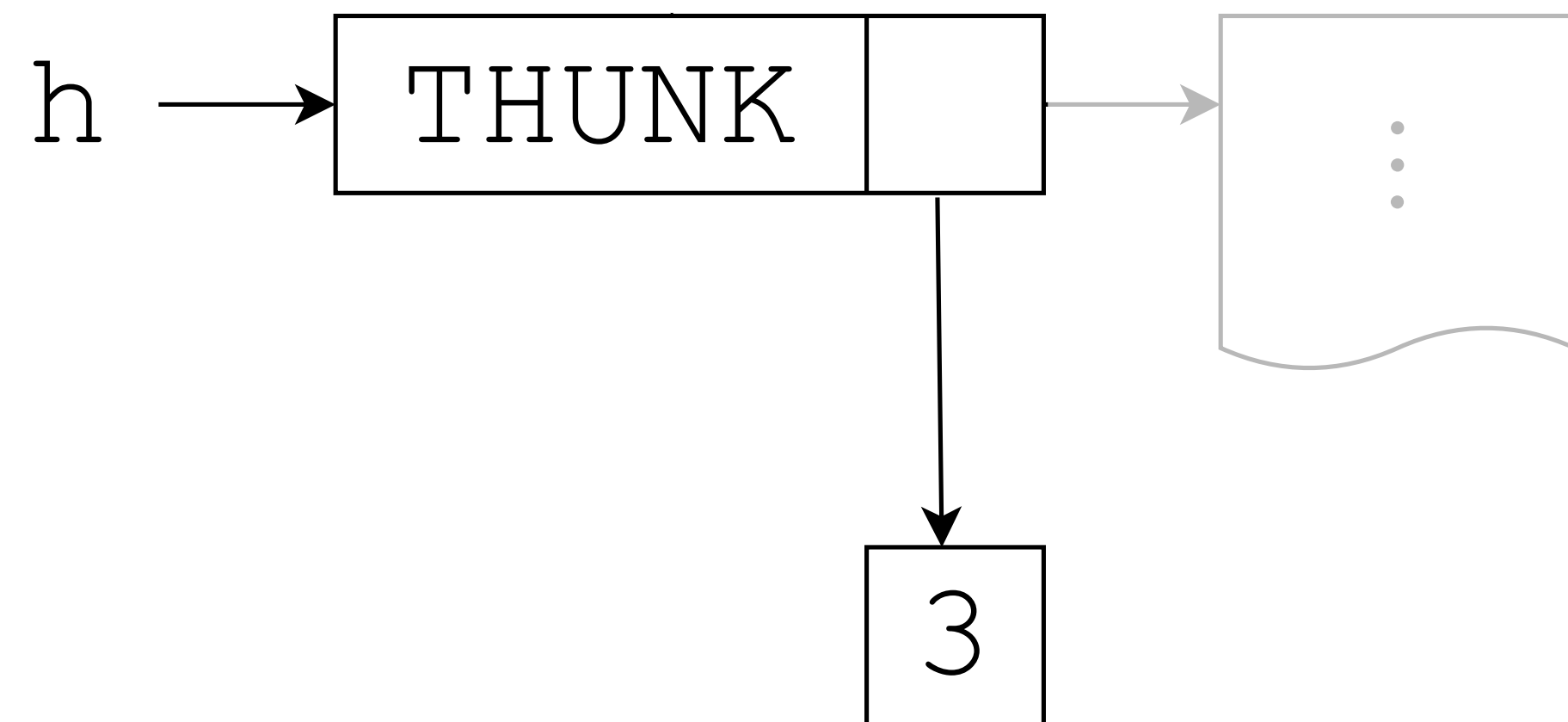


- ・サング式を評価
- ・結果への間接参照で更新

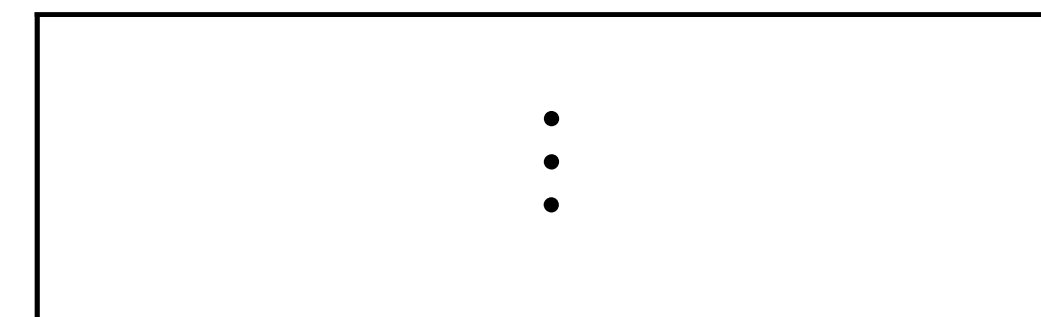
# サング評価の意味論

```
let h = THUNK( (+) 1 2) in h
```

ヒープ



スタック



- ・サング式を評価
- ・結果への間接参照で更新

# 関数適用の意味論

go 0 ones

ヒープ

0

ones → 

:	...
---	-----

go → 

pop	\$1
pop	\$2
	⋮

スタック

⋮
---

- ・関数の引数をスタックにプッシュ
- ・関数コードへジャンプ
- ・引数が足りない場合クロージャ作成

# 関数適用の意味論

`go` `0` `ones`

ヒープ

0

`ones` → 

:	...
---	-----

`go` → 

<code>pop \$1</code>
<code>pop \$2</code>
⋮

スタック

⋮

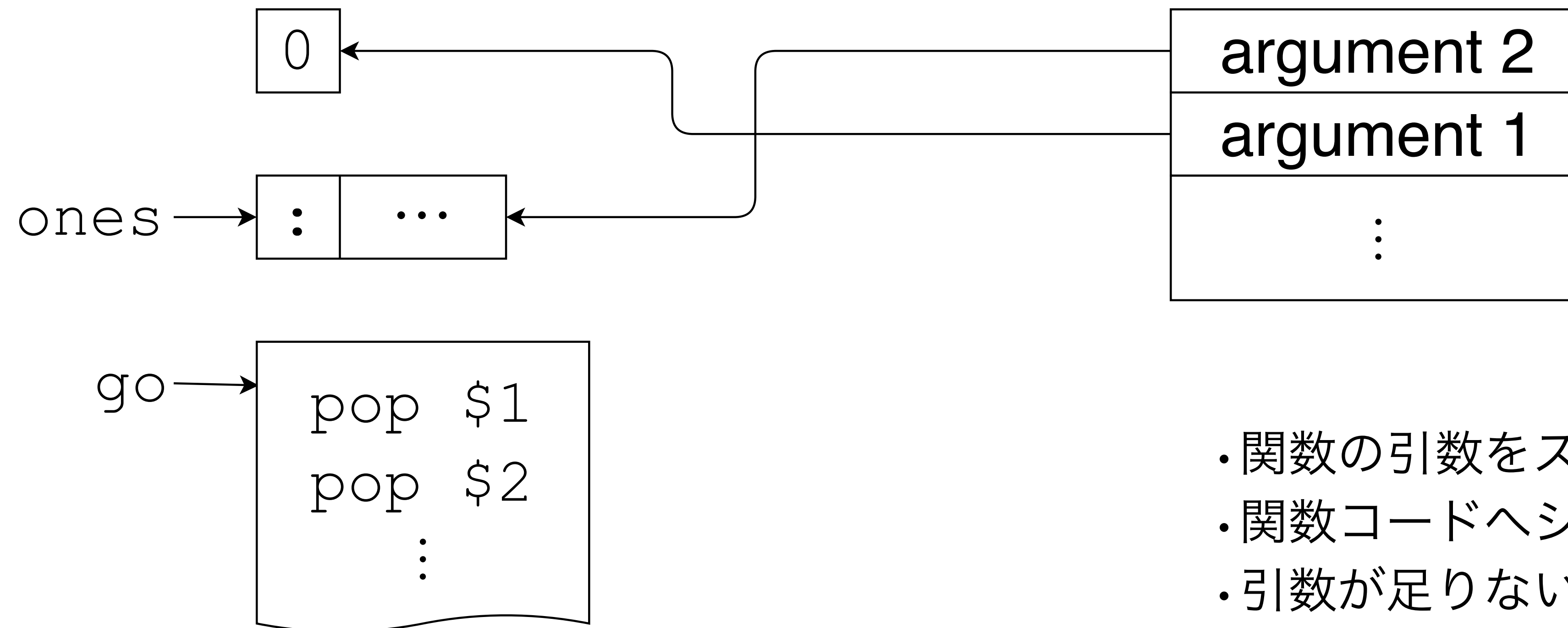
- ・関数の引数をスタックにプッシュ
- ・関数コードへジャンプ
- ・引数が足りない場合クロージャ作成

# 関数適用の意味論

go 0 ones

ヒープ

スタック



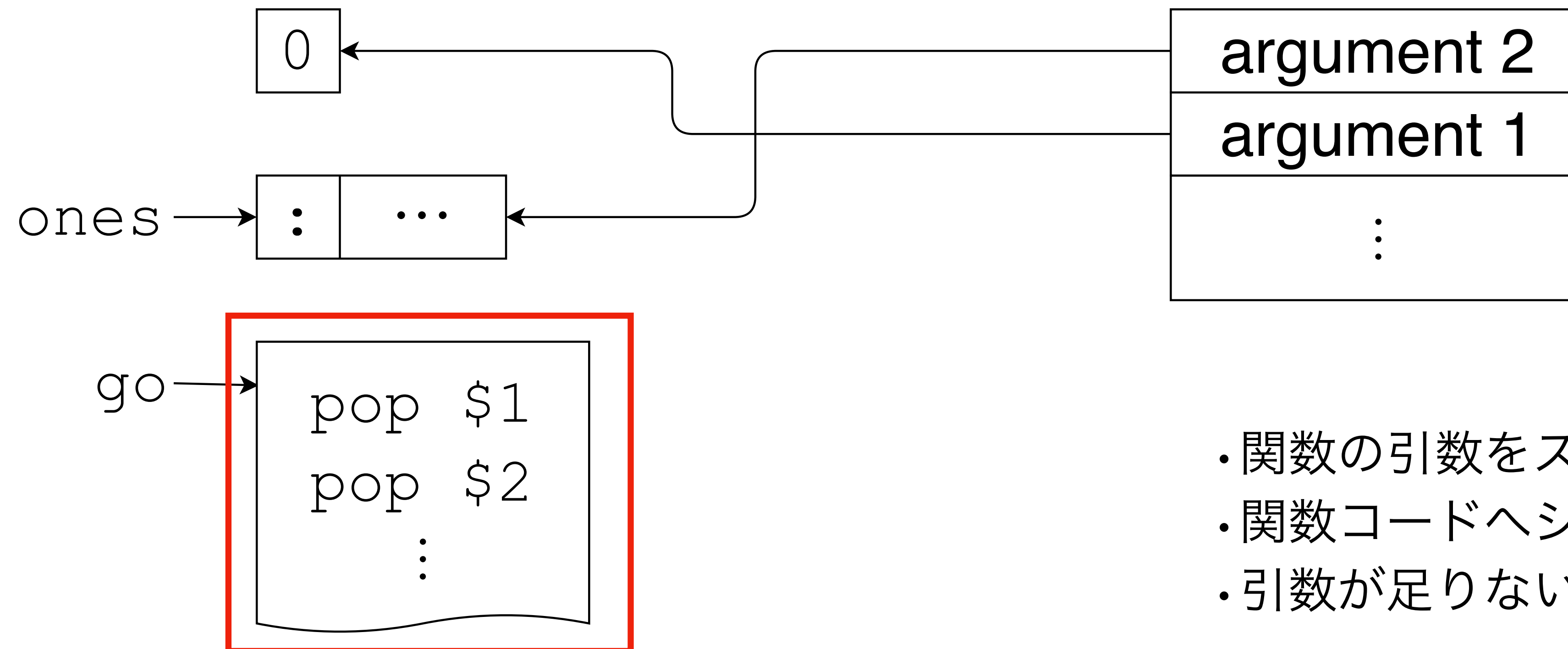
- ・関数の引数をスタックにプッシュ
- ・関数コードへジャンプ
- ・引数が足りない場合クロージャ作成

# 関数適用の意味論

go 0 ones

ヒープ

スタック



- ・関数の引数をスタックにプッシュ
- ・関数コードへジャンプ
- ・引数が足りない場合クロージャ作成



# 関数適用の意味論

go 0 ones

ヒープ

0

ones → 

:	...
---	-----

go → 

pop	\$1
pop	\$2
⋮	

スタック

⋮
---

- ・関数の引数をスタックにプッシュ
- ・関数コードへジャンプ
- ・引数が足りない場合クロージャ作成

# STG プログラムの動作

スタック

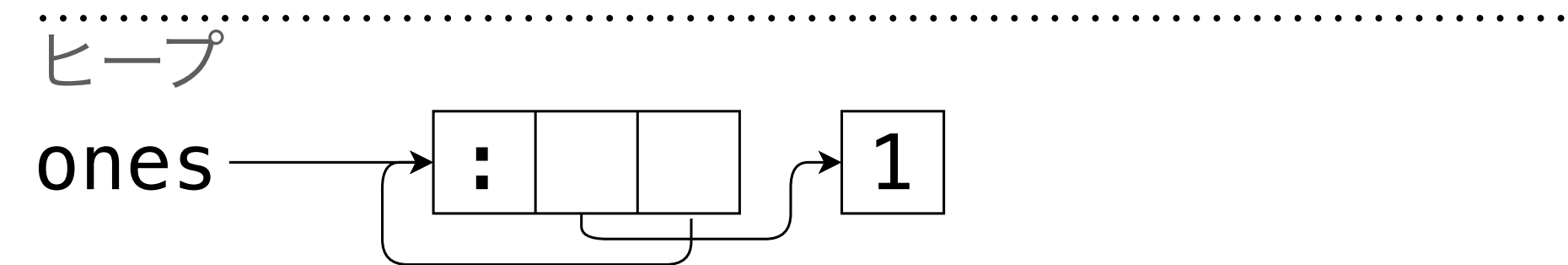
```
let ones = CON( (:) 1 ones) in
let go = FUN(z0 xs0 -> case (>=) z0 100 of {
    True -> z0;
    False -> case xs0 of {
        [] -> z0;
        (:) x xs ->
            let a1 = THUNK((+) x z0) in
            go a1 xs;
    });
}) in
let go_0_ones = THUNK(go 0 ones) in
print go_0_ones
```

.....  
ヒープ

# STG プログラムの動作

スタック

```
let ones = CON((:) 1 ones) in
let go = FUN(z0 xs0 -> case (>=) z0 100 of {
    True -> z0;
    False -> case xs0 of {
        [] -> z0;
        (:) x xs ->
            let a1 = THUNK((+) x z0) in
            go a1 xs;
    });
}) in
let go_0_ones = THUNK(go 0 ones) in
print go_0_ones
```

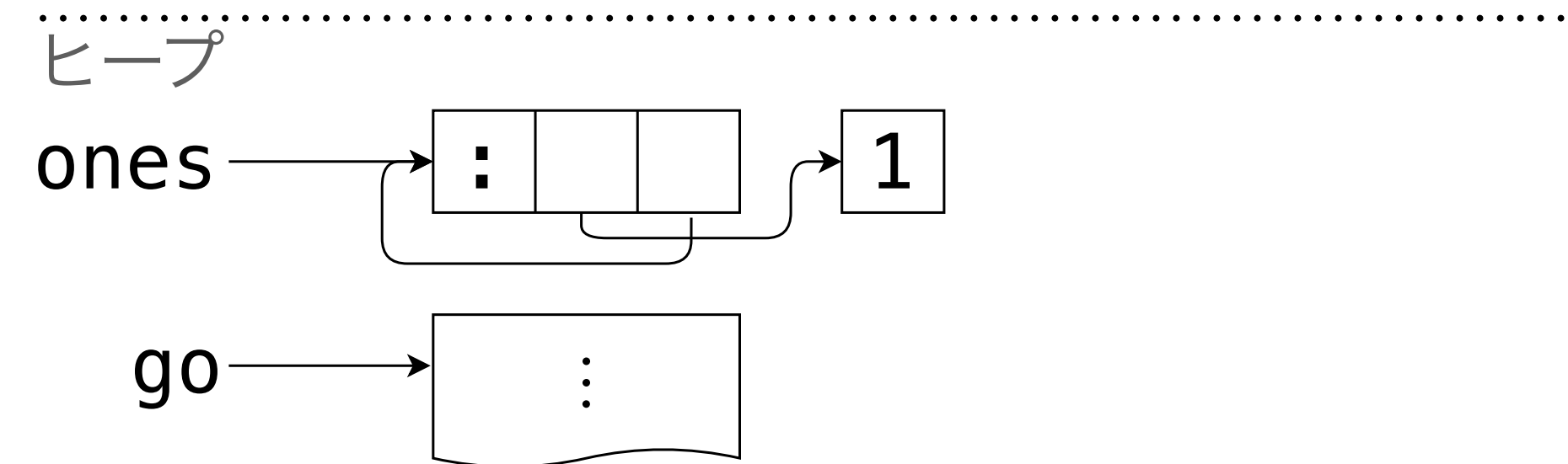


# STG プログラムの動作

スタック

```
let ones = CON( (:) 1 ones) in
let go = FUN(z0 xs0 -> case (>=) z0 100 of {
  True -> z0;
  False -> case xs0 of {
    [] -> z0;
    (:) x xs ->
      let a1 = THUNK((+) x z0) in
      go a1 xs;
  });
  }) in
```

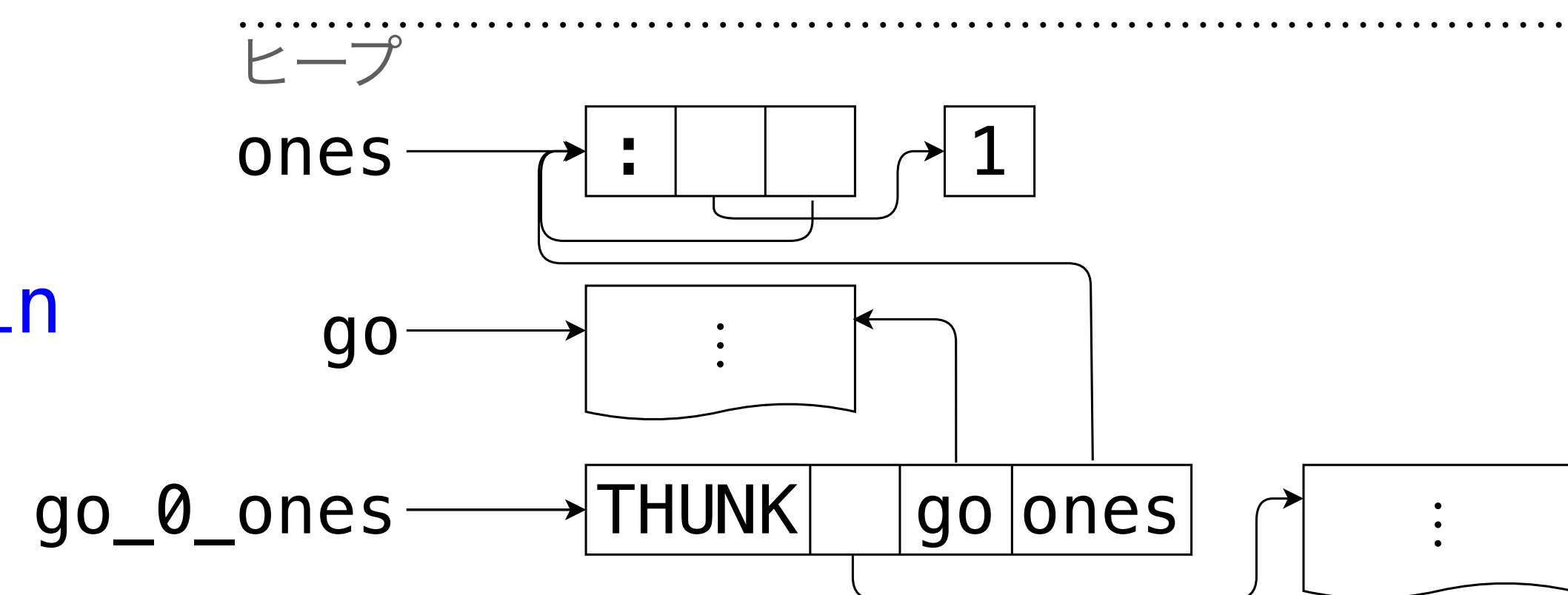
```
let go_0_ones = THUNK(go 0 ones) in
print go_0_ones
```



# STG プログラムの動作

スタック

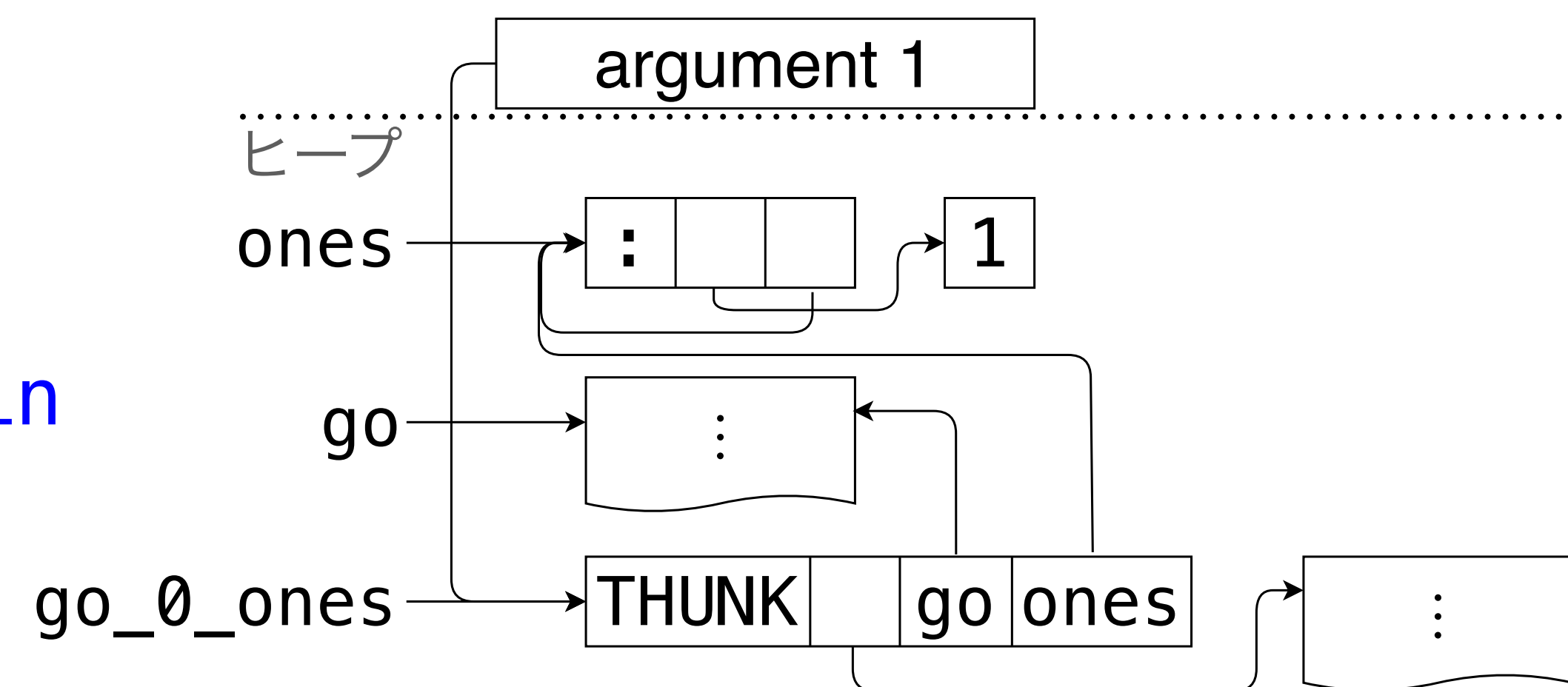
```
let ones = CON( (:) 1 ones) in
let go = FUN(z0 xs0 -> case (>=) z0 100 of {
    True -> z0;
    False -> case xs0 of {
        [] -> z0;
        (:) x xs ->
            let a1 = THUNK((+) x z0) in
            go a1 xs;
    };
}) in
let go_0_ones = THUNK(go 0 ones) in
print go_0_ones
```



# STG プログラムの動作

スタック

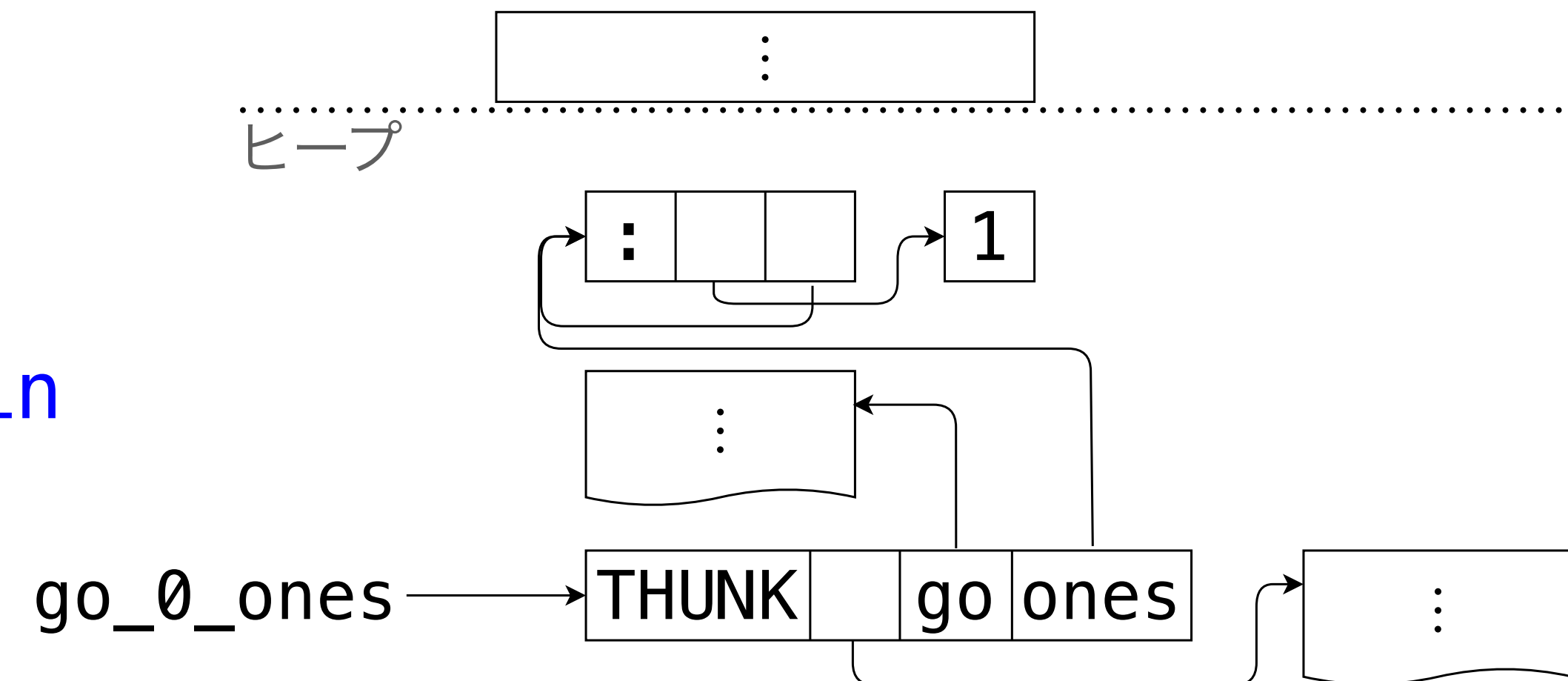
```
let ones = CON( (:) 1 ones) in
let go = FUN(z0 xs0 -> case (>=) z0 100 of {
    True -> z0;
    False -> case xs0 of {
        [] -> z0;
        (:) x xs ->
            let a1 = THUNK((+) x z0) in
            go a1 xs;
    };
}) in
let go_0_ones = THUNK(go 0 ones) in
print go_0_ones
```



# STG プログラムの動作

スタック

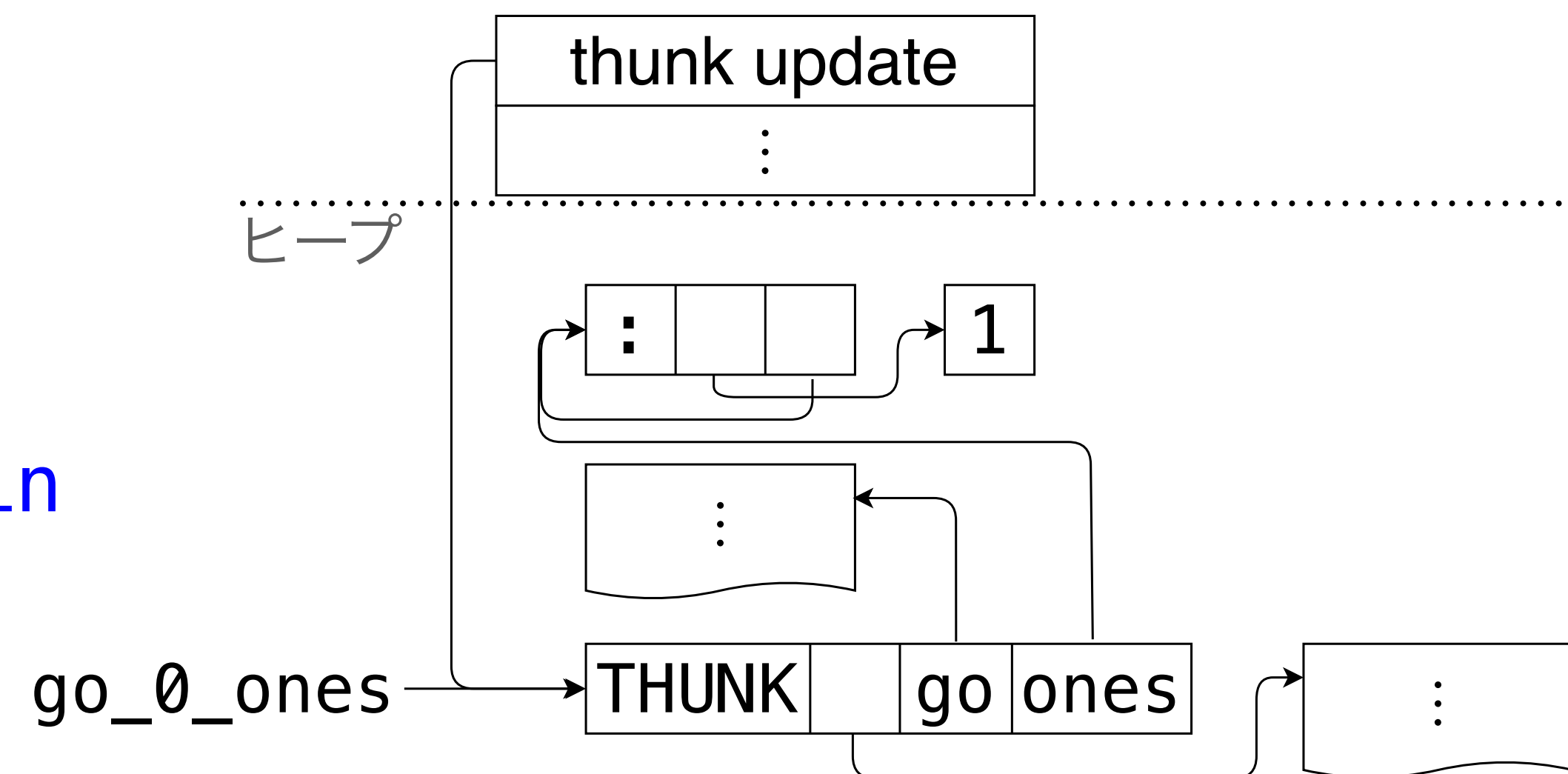
```
let ones = CON( (:) 1 ones) in
let go = FUN(z0 xs0 -> case (>=) z0 100 of {
  True -> z0;
  False -> case xs0 of {
    [] -> z0;
    (:) x xs ->
      let a1 = THUNK((+) x z0) in
      go a1 xs;
  });
}) in
let go_0_ones = THUNK(go 0 ones) in
print go_0_ones
```



# STG プログラムの動作

スタック

```
let ones = CON( (:) 1 ones) in
let go = FUN(z0 xs0 -> case (>=) z0 100 of {
  True -> z0;
  False -> case xs0 of {
    [] -> z0;
    (:) x xs ->
      let a1 = THUNK((+) x z0) in
      go a1 xs;
  });
}) in
let go_0_ones = THUNK(go 0 ones) in
print go_0_ones
```

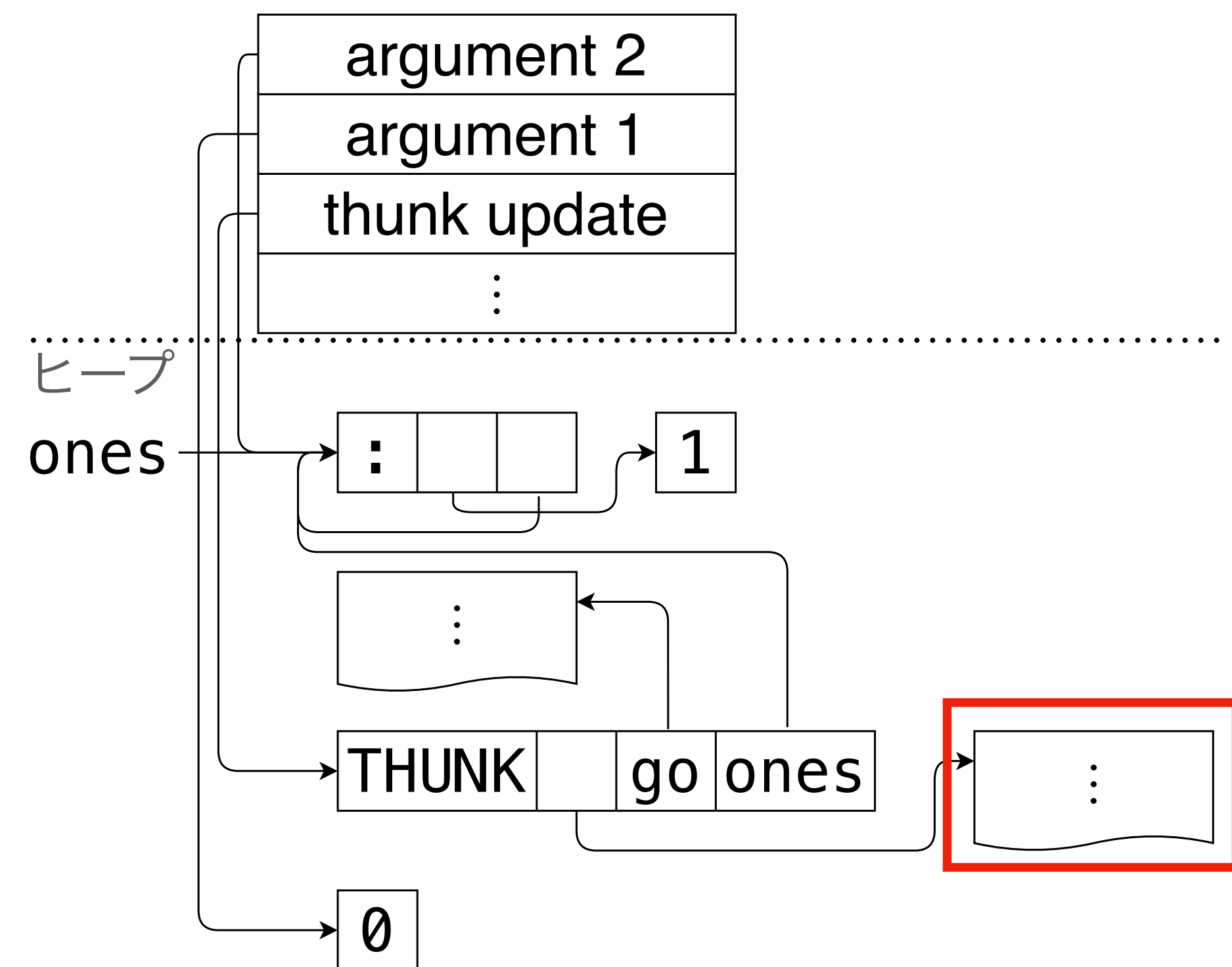




# STG プログラムの動作

スタック

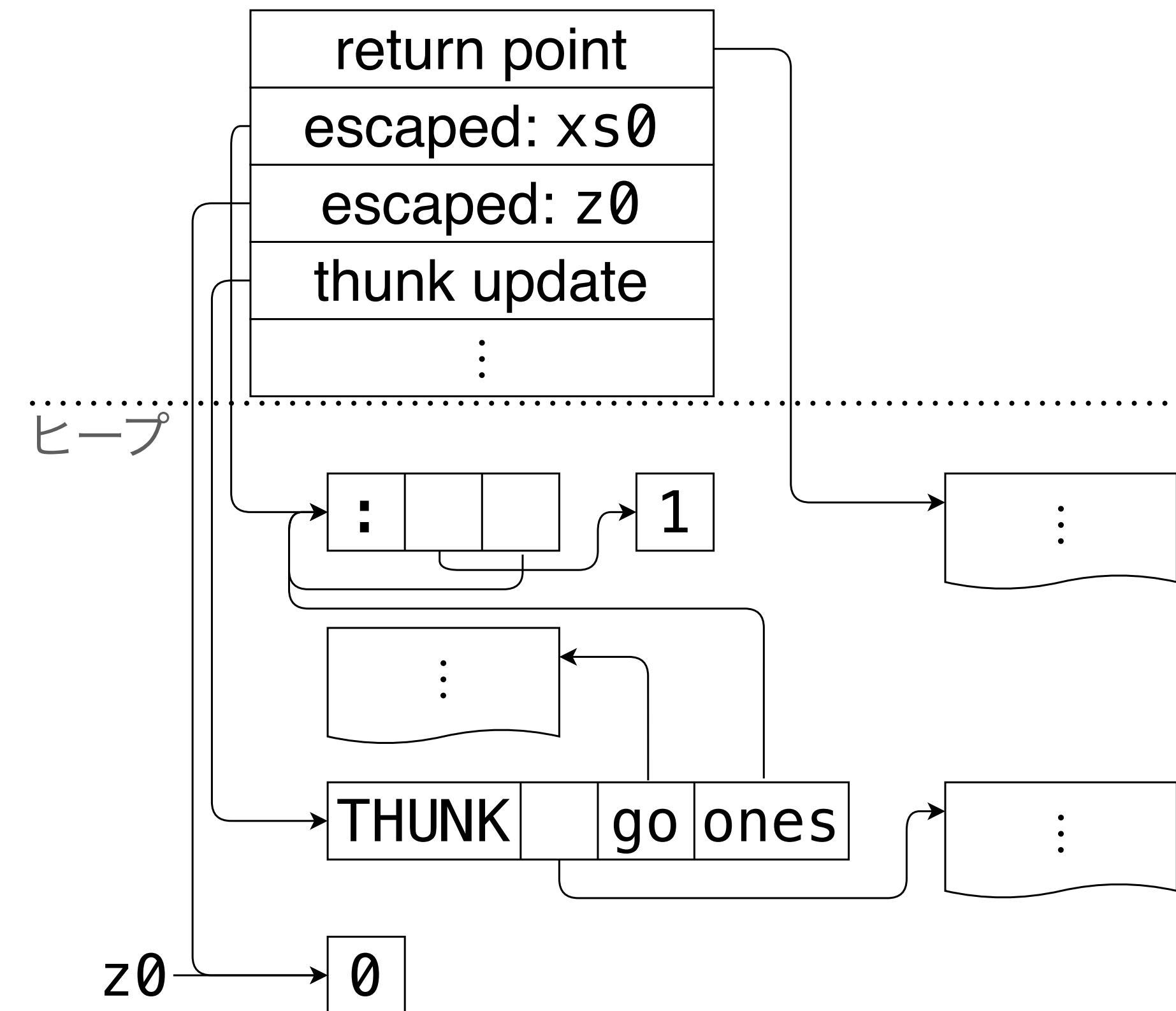
```
let ones = CON( (:) 1 ones) in
let go = FUN(z0 xs0 -> case (>=) z0 100 of {
  True -> z0;
  False -> case xs0 of {
    [] -> z0;
    (:) x xs ->
      let a1 = THUNK((+) x z0) in
      go a1 xs;
  });
}) in
let go_0_ones = THUNK(go 0 ones) in
print go_0_ones
```



# STG プログラムの動作

スタック

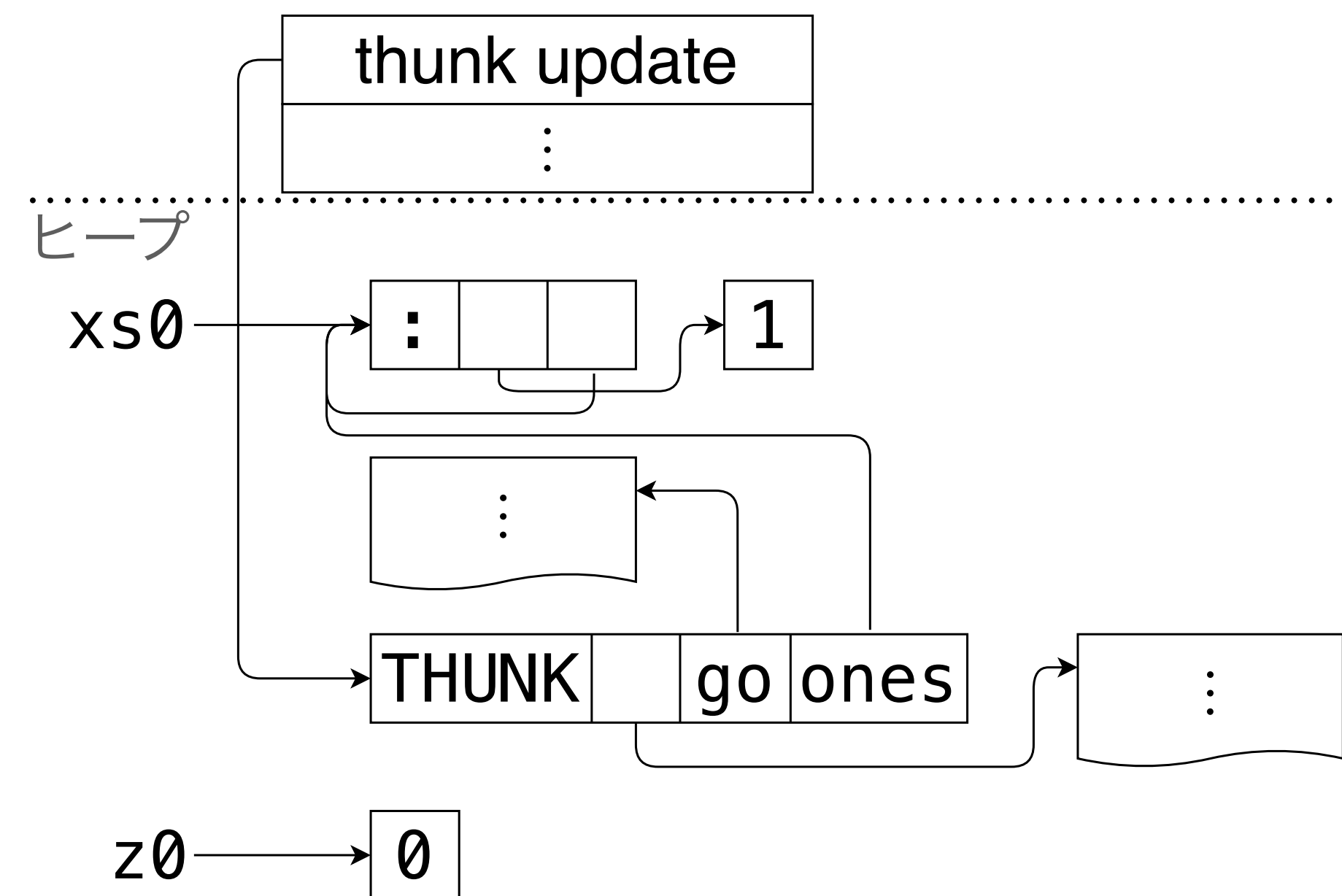
```
let ones = CON( (:) 1 ones) in
let go = FUN(z0 xs0 -> case (>=) z0 100 of {
    True -> z0;
    False -> case xs0 of {
        [] -> z0;
        (:) x xs ->
            let a1 = THUNK((+) x z0) in
            go a1 xs;
    };
}) in
let go_0_ones = THUNK(go 0 ones) in
print go_0_ones
```



# STG プログラムの動作

スタック

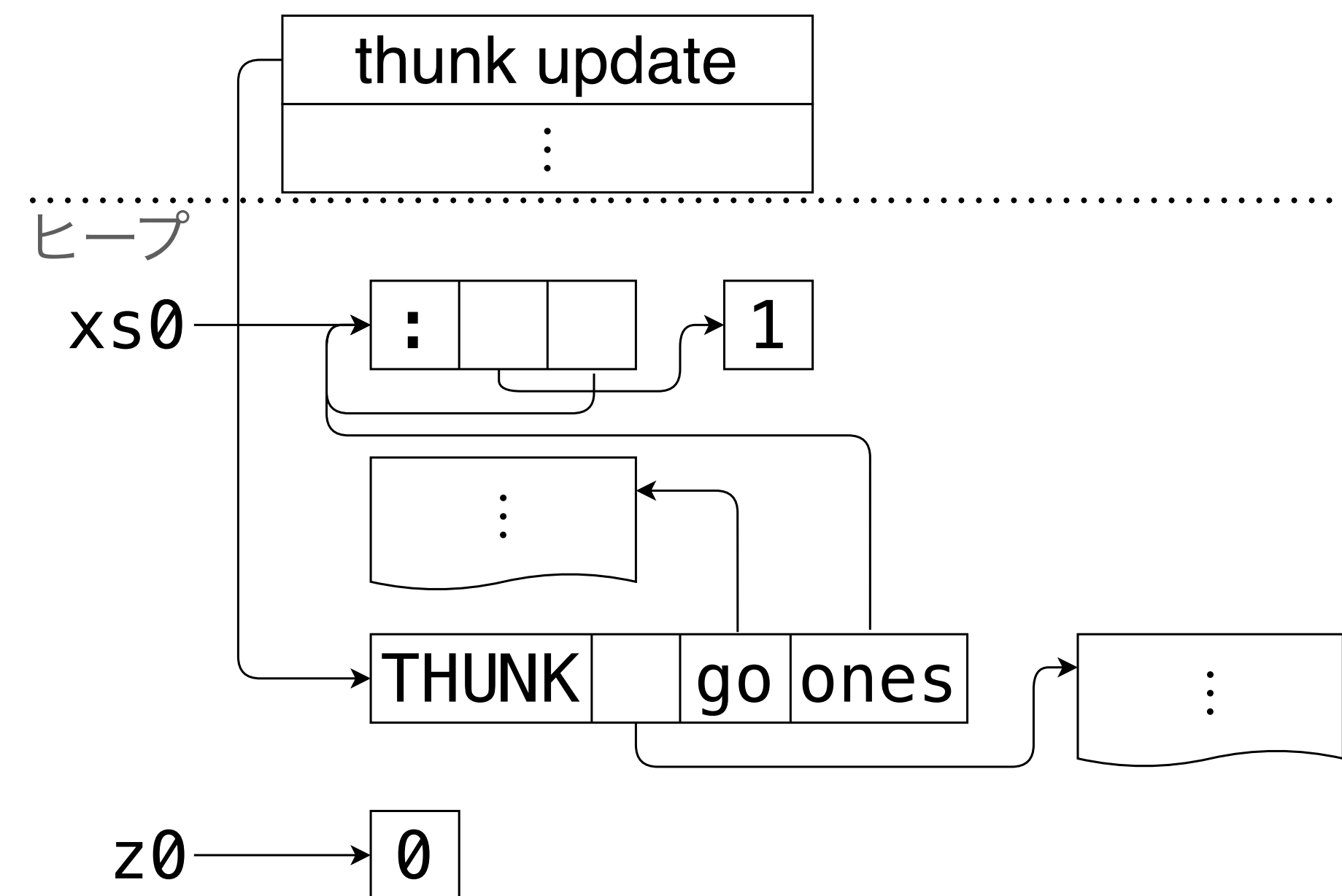
```
let ones = CON( (:) 1 ones) in
let go = FUN(z0 xs0 -> case (>=) z0 100 of {
    True -> z0;
    False -> case xs0 of {
        [] -> z0;
        (:) x xs ->
            let a1 = THUNK((+) x z0) in
            go a1 xs;
    };
}) in
let go_0_ones = THUNK(go 0 ones) in
print go_0_ones
```



# STG プログラムの動作

スタック

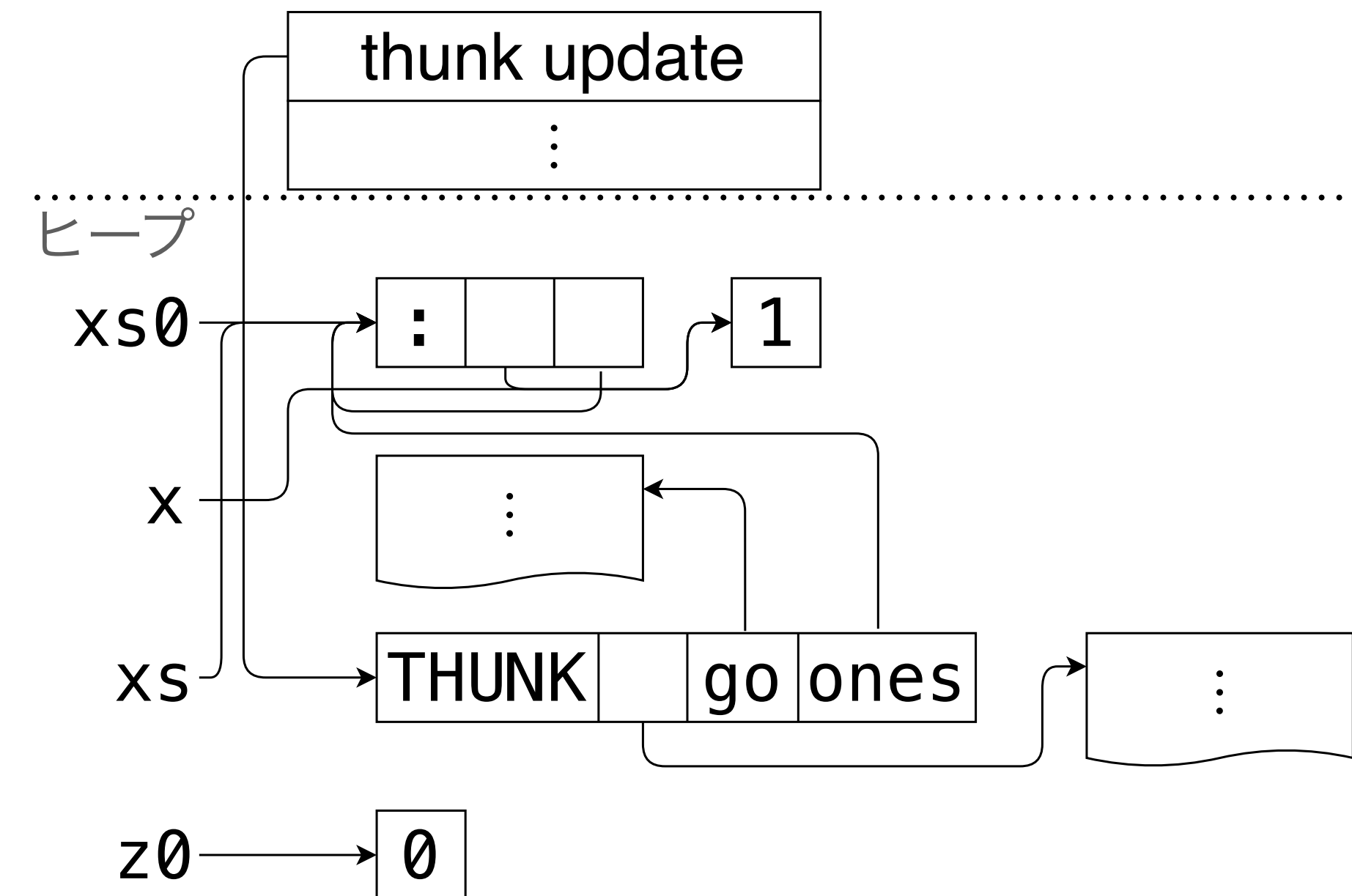
```
let ones = CON( (:) 1 ones) in
let go = FUN(z0 xs0 -> case (>=) z0 100 of {
    True -> z0;
    False -> case xs0 of {
        [] -> z0;
        (:) x xs ->
            let a1 = THUNK((+) x z0) in
            go a1 xs;
    };
}) in
let go_0_ones = THUNK(go 0 ones) in
print go_0_ones
```



# STG プログラムの動作

スタック

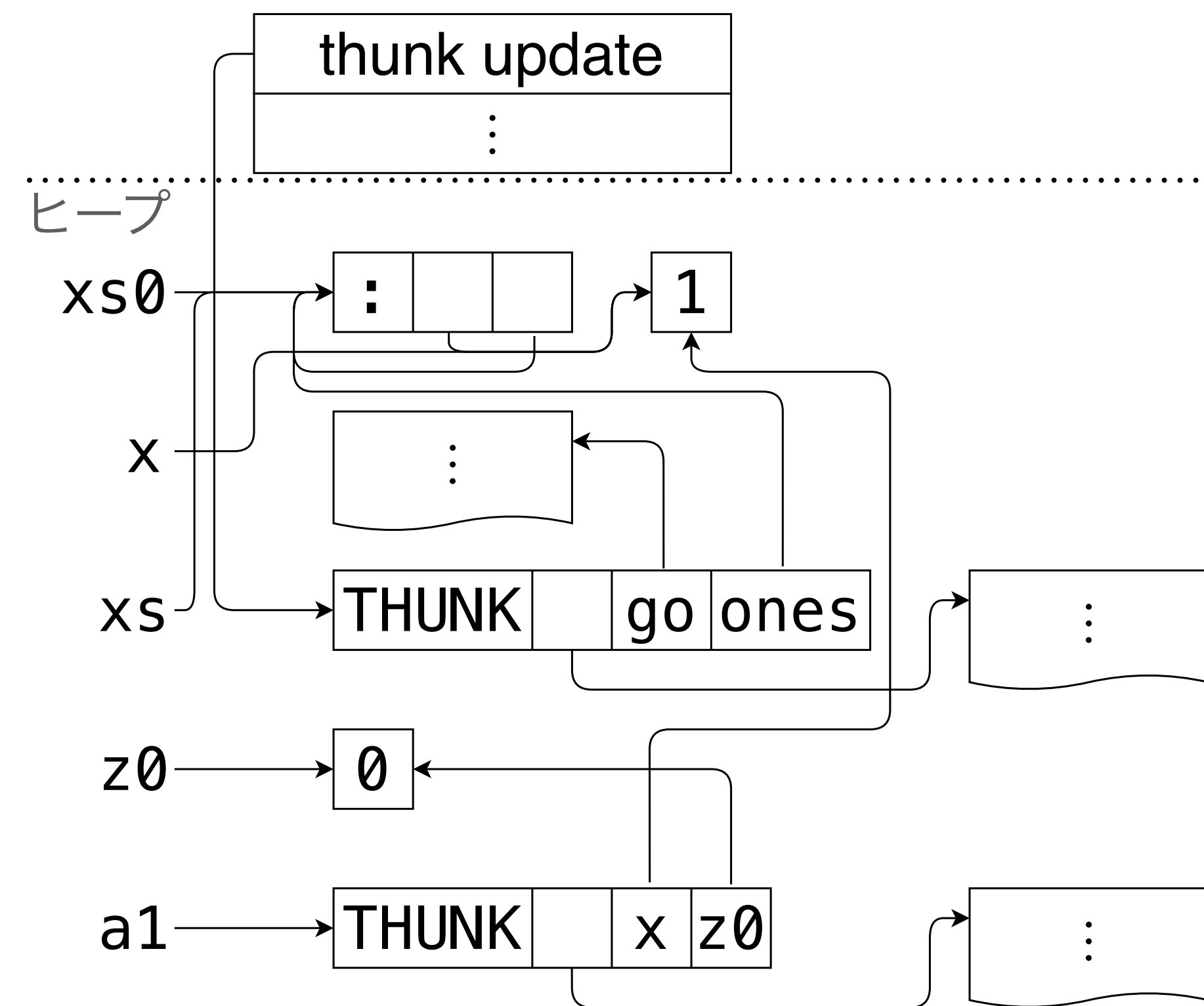
```
let ones = CON( (:) 1 ones) in
let go = FUN(z0 xs0 -> case (>=) z0 100 of {
  True -> z0;
  False -> case xs0 of {
    [] -> z0;
    (:) x xs ->
      let a1 = THUNK((+) x z0) in
      go a1 xs;
  };
}) in
let go_0_ones = THUNK(go 0 ones) in
print go_0_ones
```



# STG プログラムの動作

スタック

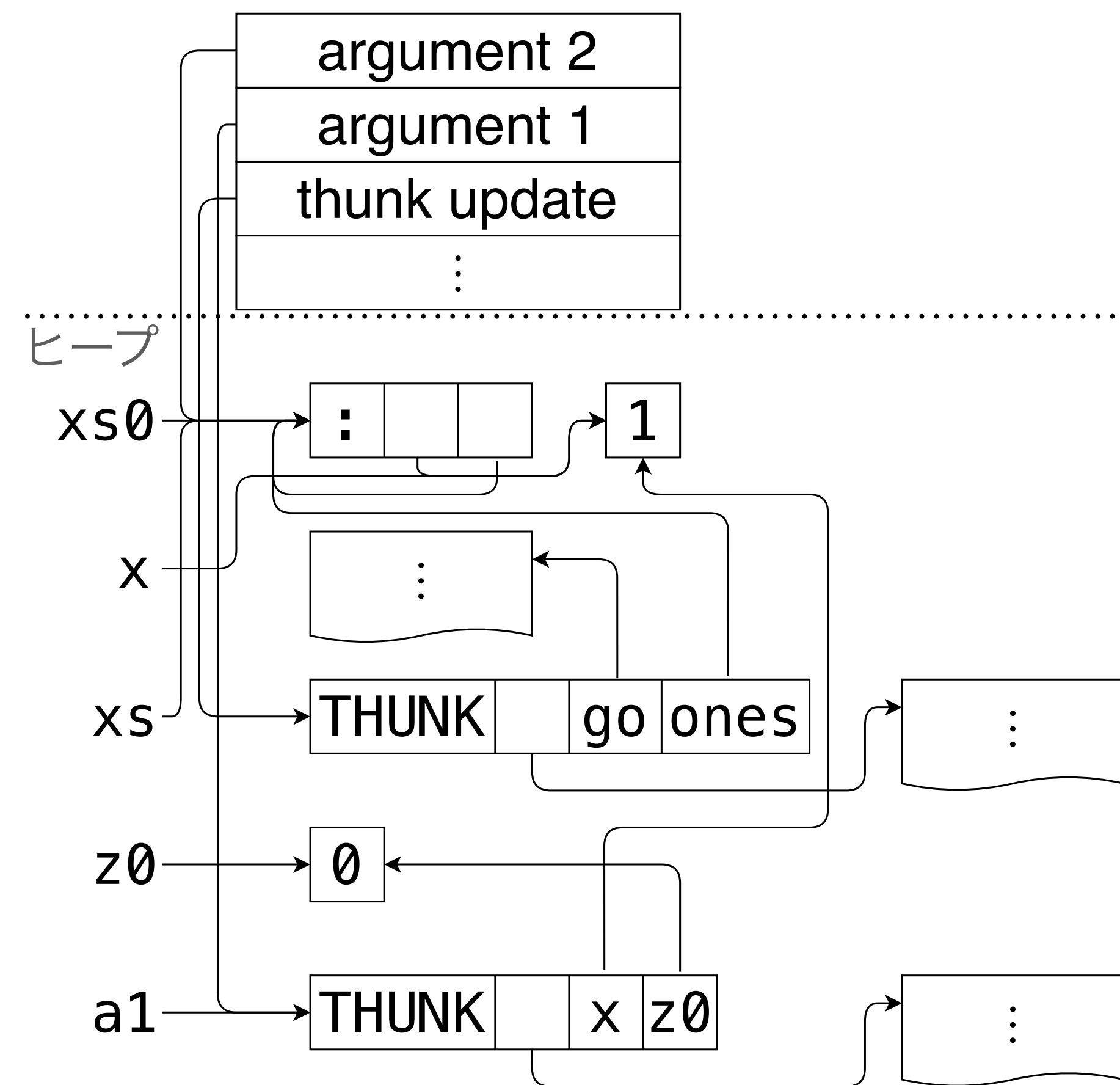
```
let ones = CON( (:) 1 ones) in
let go = FUN(z0 xs0 -> case (>=) z0 100 of {
  True -> z0;
  False -> case xs0 of {
    [] -> z0;
    (:) x xs ->
      let a1 = THUNK((+) x z0) in
      go a1 xs;
  };
}) in
let go_0_ones = THUNK(go 0 ones) in
print go_0_ones
```



# STG プログラムの動作

スタック

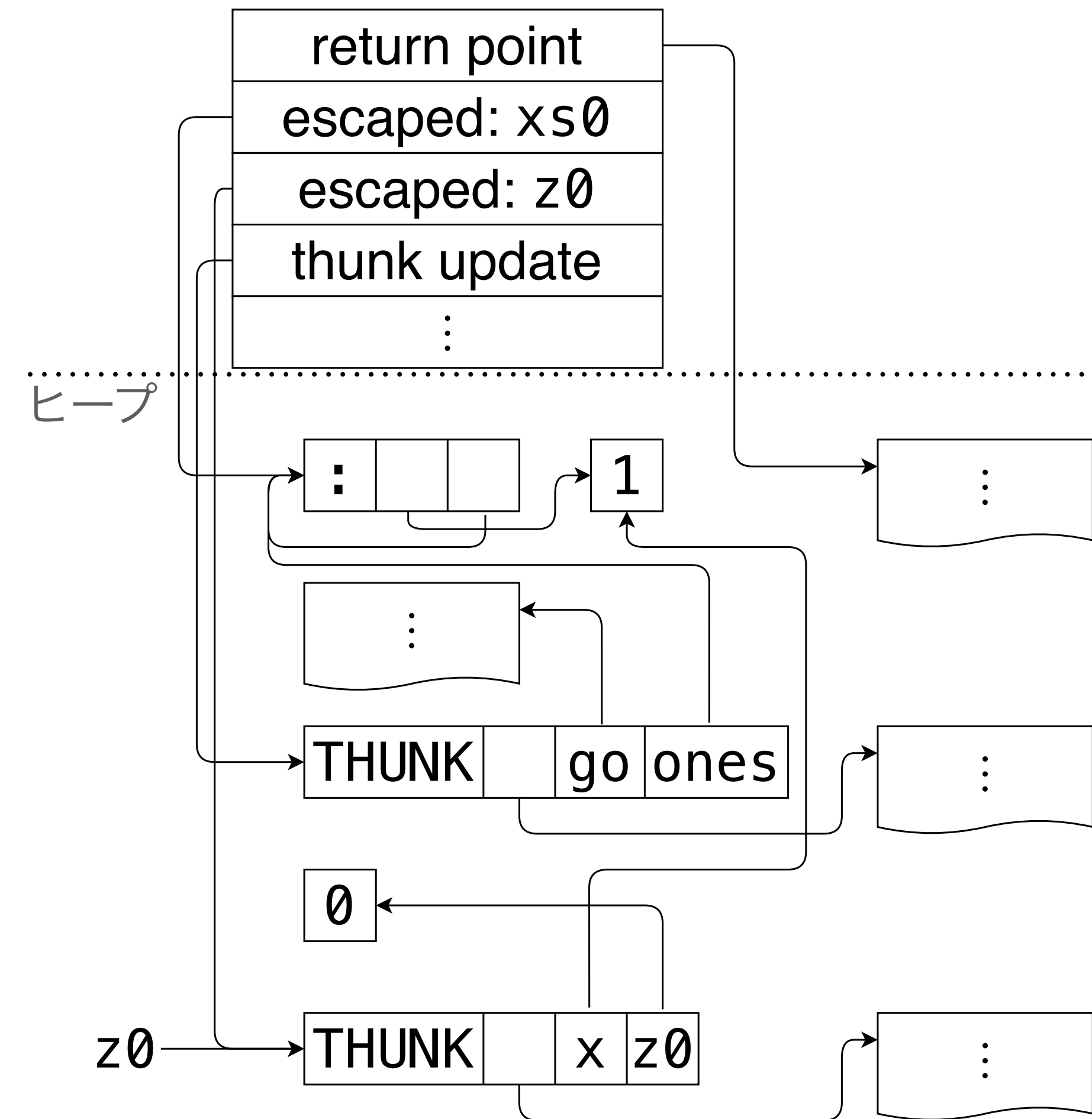
```
let ones = CON( (:) 1 ones) in
let go = FUN(z0 xs0 -> case (>=) z0 100 of {
    True -> z0;
    False -> case xs0 of {
        [] -> z0;
        (:) x xs ->
            let a1 = THUNK((+) x z0) in
            go a1 xs;
    };
}) in
let go_0_ones = THUNK(go 0 ones) in
print go_0_ones
```



# STG プログラムの動作

スタック

```
let ones = CON( (:) 1 ones) in
let go = FUN(z0 xs0 -> case (>=) z0 100 of {
    True -> z0;
    False -> case xs0 of {
        [] -> z0;
        (:) x xs ->
            let a1 = THUNK((+) x z0) in
            go a1 xs;
    };
}) in
let go_0_ones = THUNK(go 0 ones) in
print go_0_ones
```

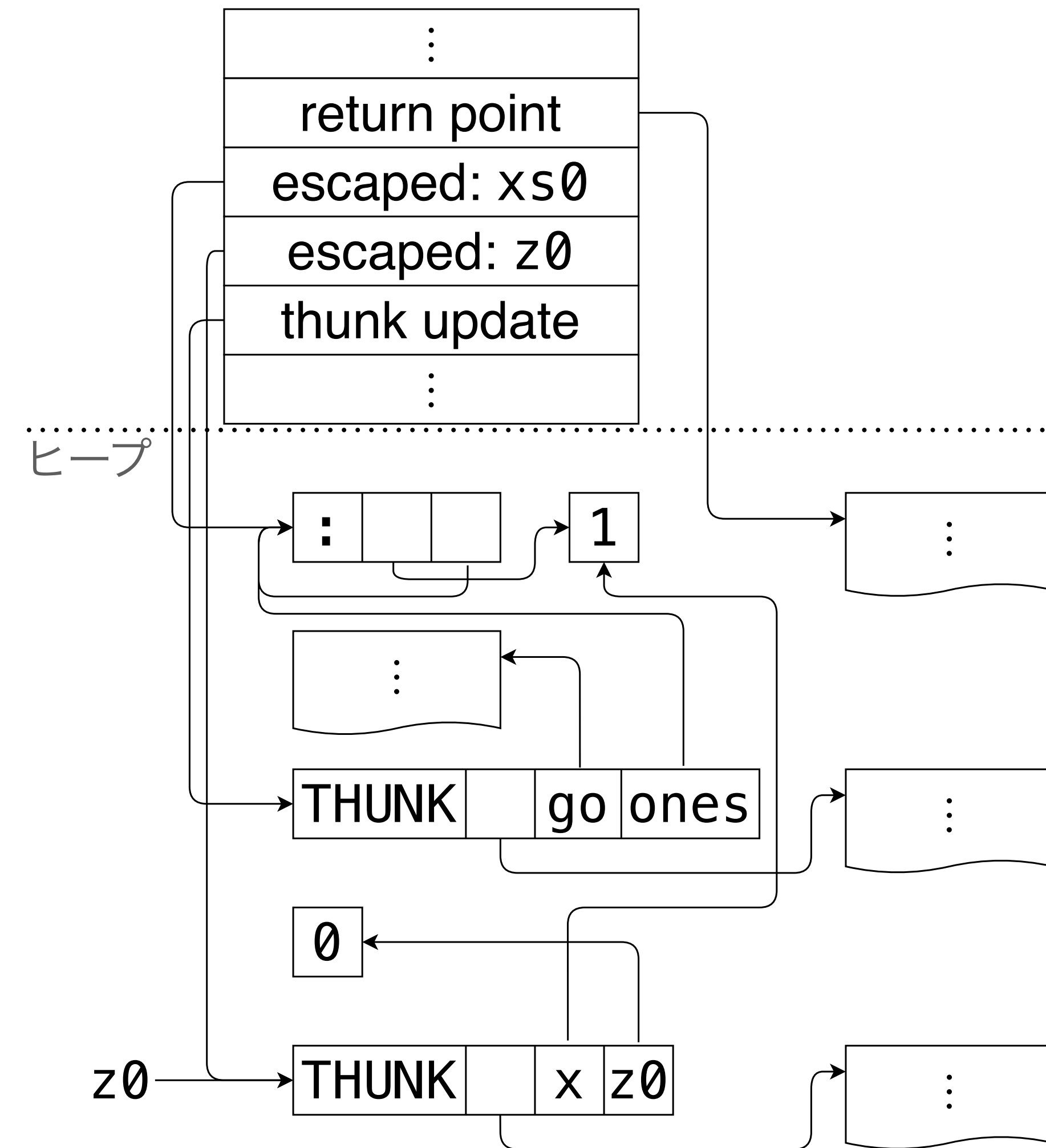




# STG プログラムの動作

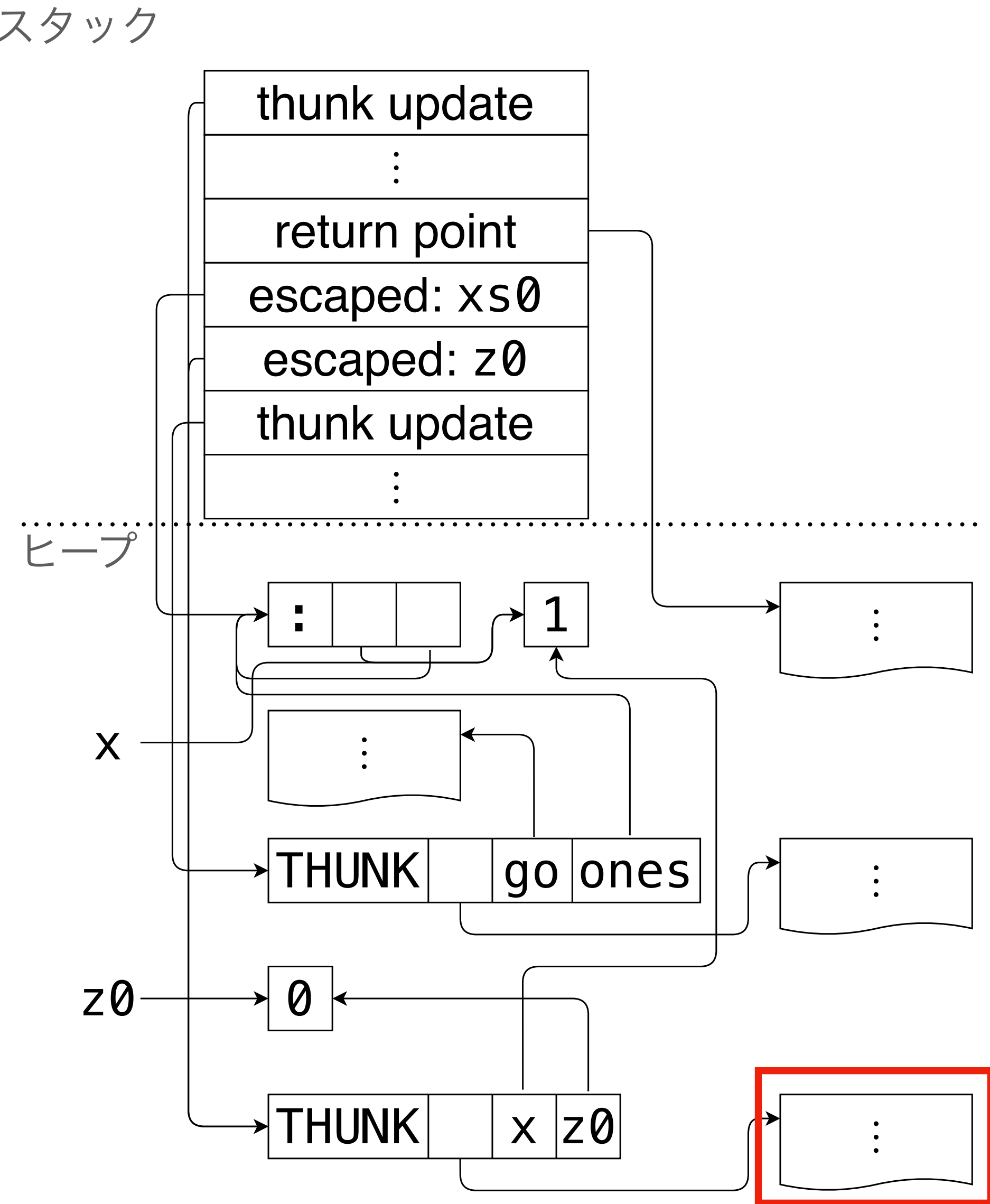
```
let ones = CON( (:) 1 ones) in
let go = FUN(z0 xs0 -> case (>=) z0 100 of {
  True -> z0;
  False -> case xs0 of {
    [] -> z0;
    (:) x xs ->
      let a1 = THUNK((+) x z0) in
      go a1 xs;
  };
}) in
let go_0_ones = THUNK(go 0 ones) in
print go_0_ones
```

スタック



# STG プログラムの動作

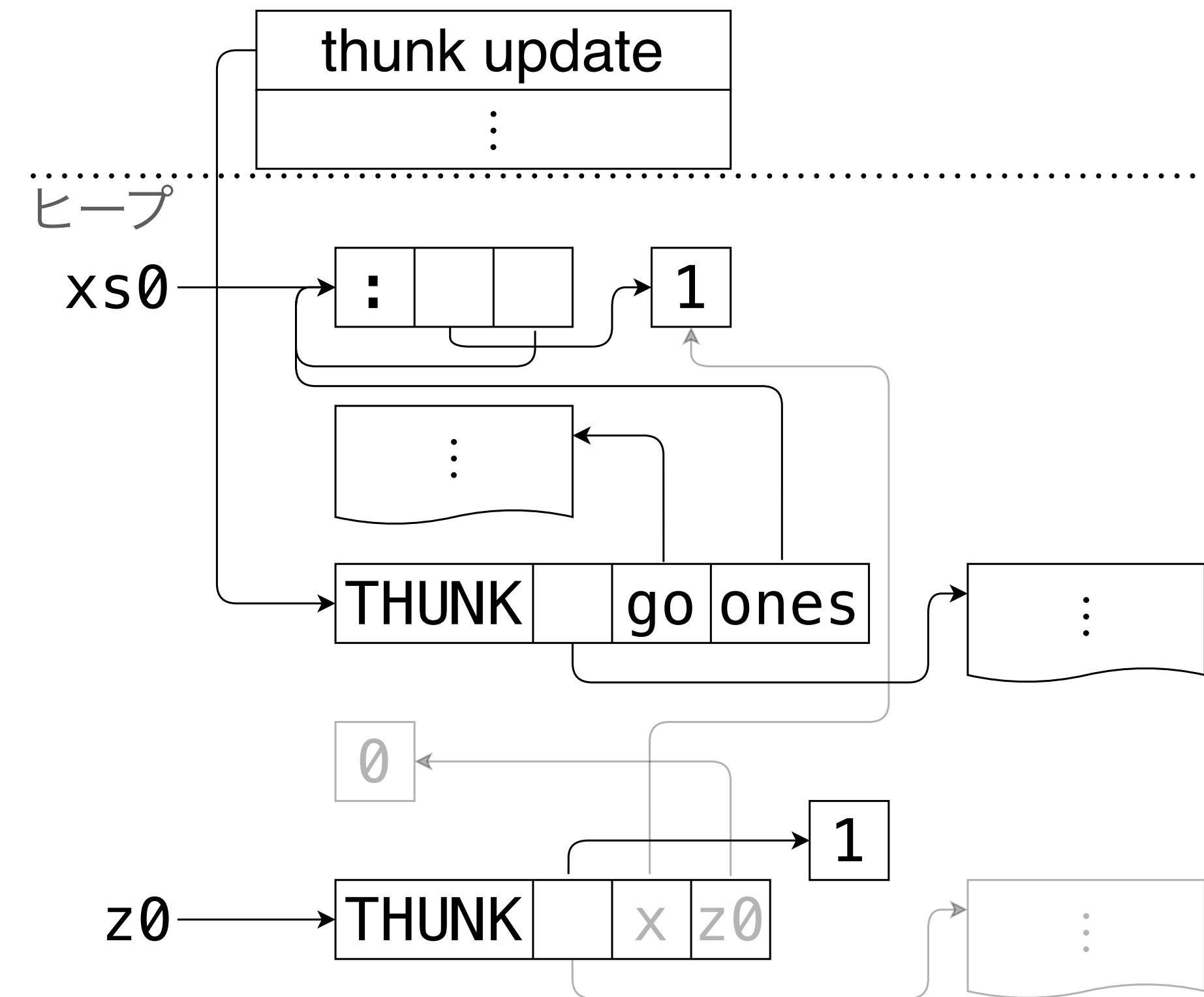
```
let ones = CON((:) 1 ones) in
let go = FUN(z0 xs0 -> case (>=) z0 100 of {
    True -> z0;
    False -> case xs0 of {
        [] -> z0;
        (:) x xs ->
            let a1 = THUNK((+) x z0) in
            go a1 xs;
    });
}) in
let go_0_ones = THUNK(go 0 ones) in
print go_0_ones
```



# STG プログラムの動作

スタック

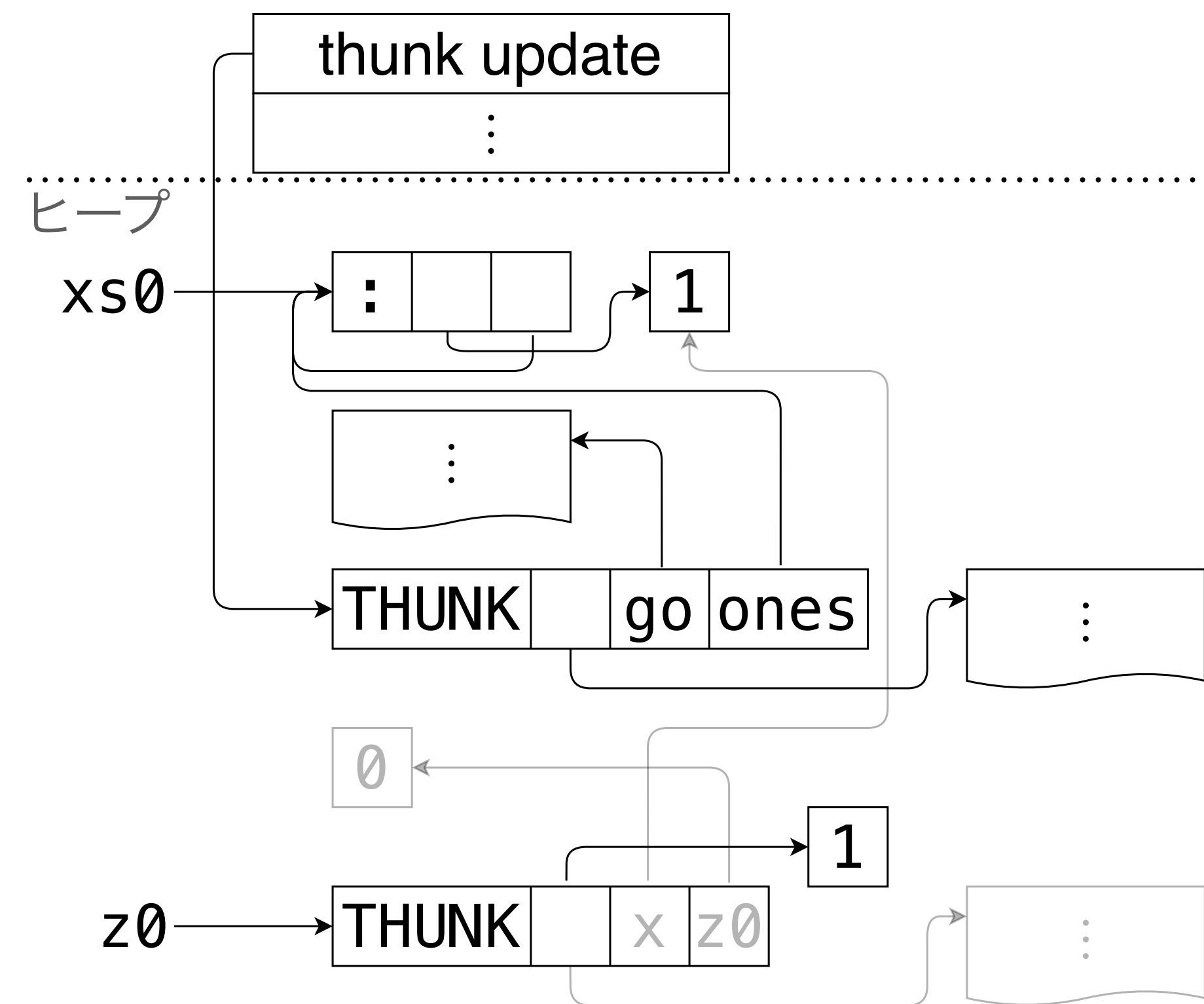
```
let ones = CON( (:) 1 ones) in
let go = FUN(z0 xs0 -> case (>=) z0 100 of {
    True -> z0;
    False -> case xs0 of {
        [] -> z0;
        (:) x xs ->
            let a1 = THUNK((+) x z0) in
            go a1 xs;
    };
}) in
let go_0_ones = THUNK(go 0 ones) in
print go_0_ones
```



# STG プログラムの動作

スタック

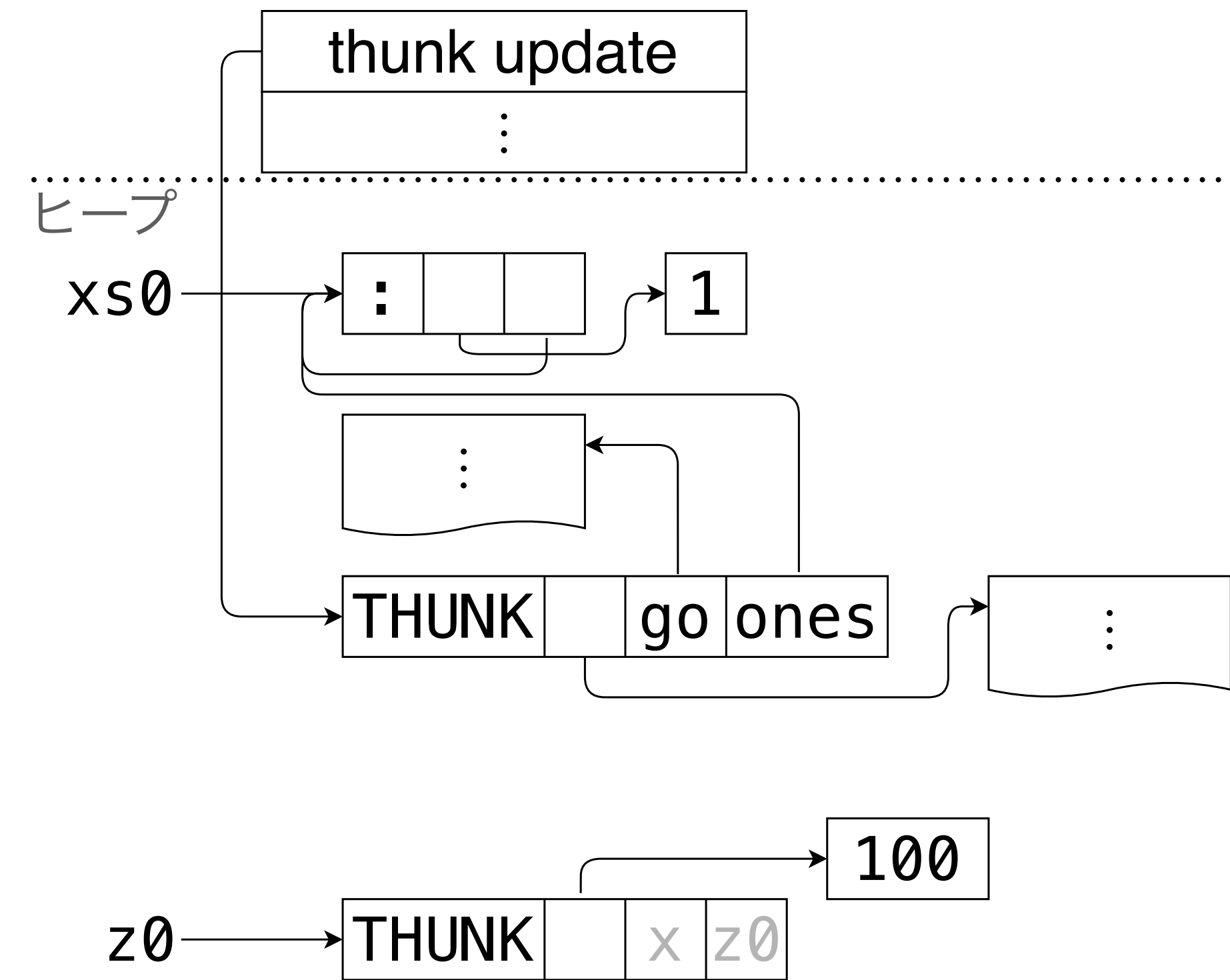
```
let ones = CON( (:) 1 ones) in
let go = FUN(z0 xs0 -> case (>=) z0 100 of {
  True -> z0;
  False -> case xs0 of {
    [] -> z0;
    (:) x xs ->
      let a1 = THUNK((+) x z0) in
      go a1 xs;
  };
}) in
let go_0_ones = THUNK(go 0 ones) in
print go_0_ones
```



# STG プログラムの動作

スタック

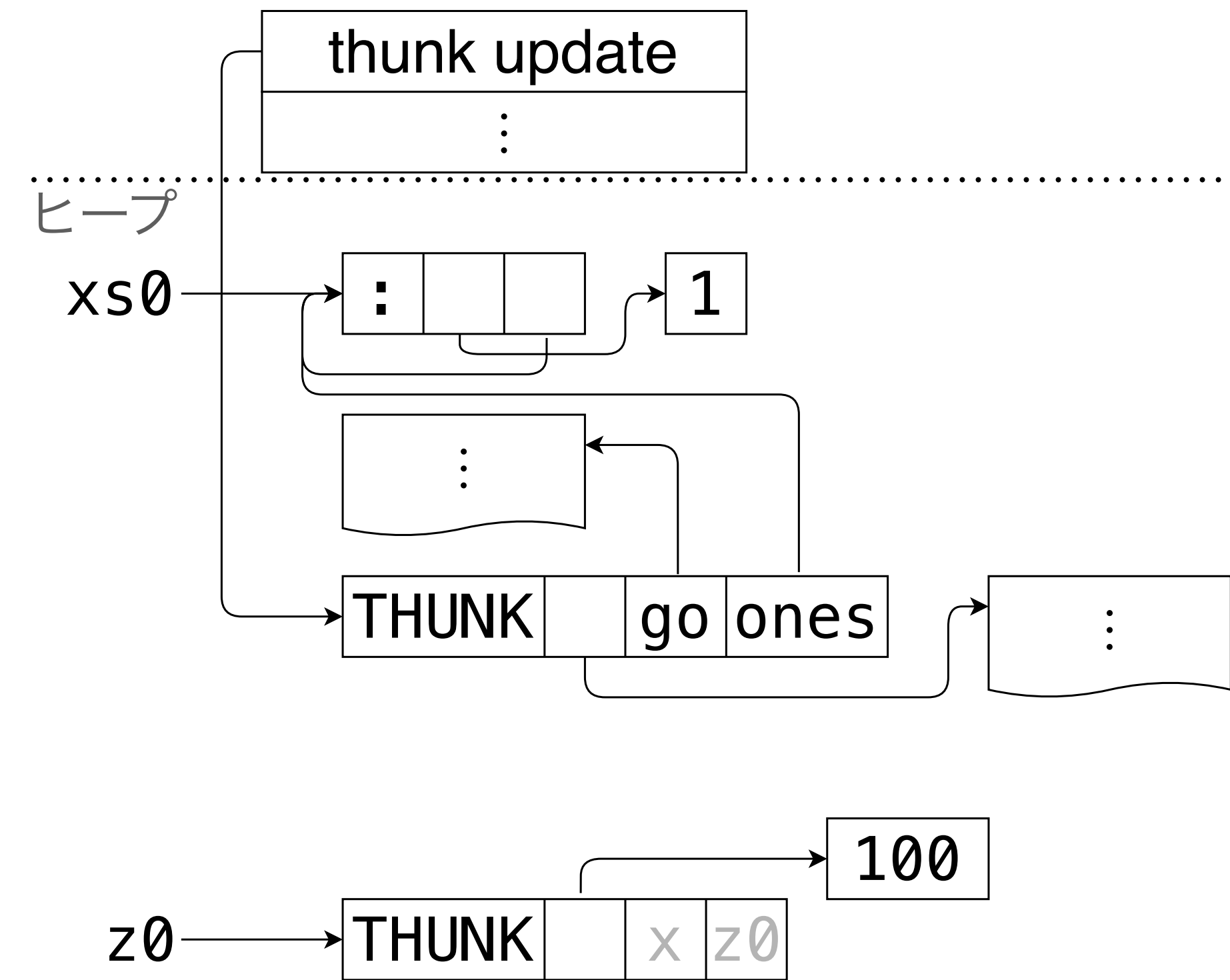
```
let ones = CON( (:) 1 ones) in
let go = FUN(z0 xs0 -> case (>=) z0 100 of {
    True -> z0;
    False -> case xs0 of {
        [] -> z0;
        (:) x xs ->
            let a1 = THUNK((+) x z0) in
            go a1 xs;
    };
}) in
let go_0_ones = THUNK(go 0 ones) in
print go_0_ones
```



# STG プログラムの動作

スタック

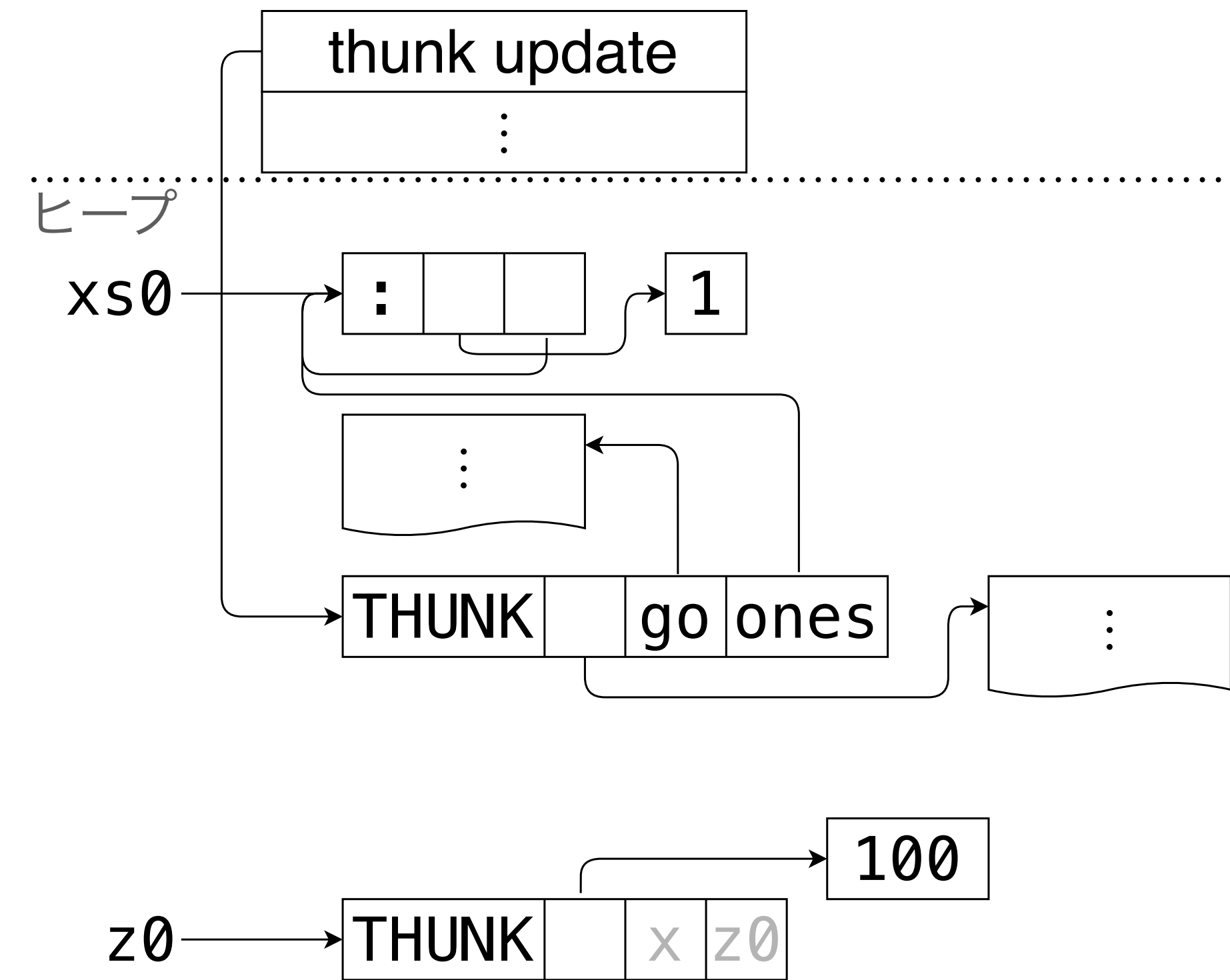
```
let ones = CON( (:) 1 ones) in
let go = FUN(z0 xs0 -> case (>=) z0 100 of {
  True -> z0;
  False -> case xs0 of {
    [] -> z0;
    (:) x xs ->
      let a1 = THUNK((+) x z0) in
      go a1 xs;
  });
in
let go_0_ones = THUNK(go 0 ones) in
print go_0_ones
```



# STG プログラムの動作

スタック

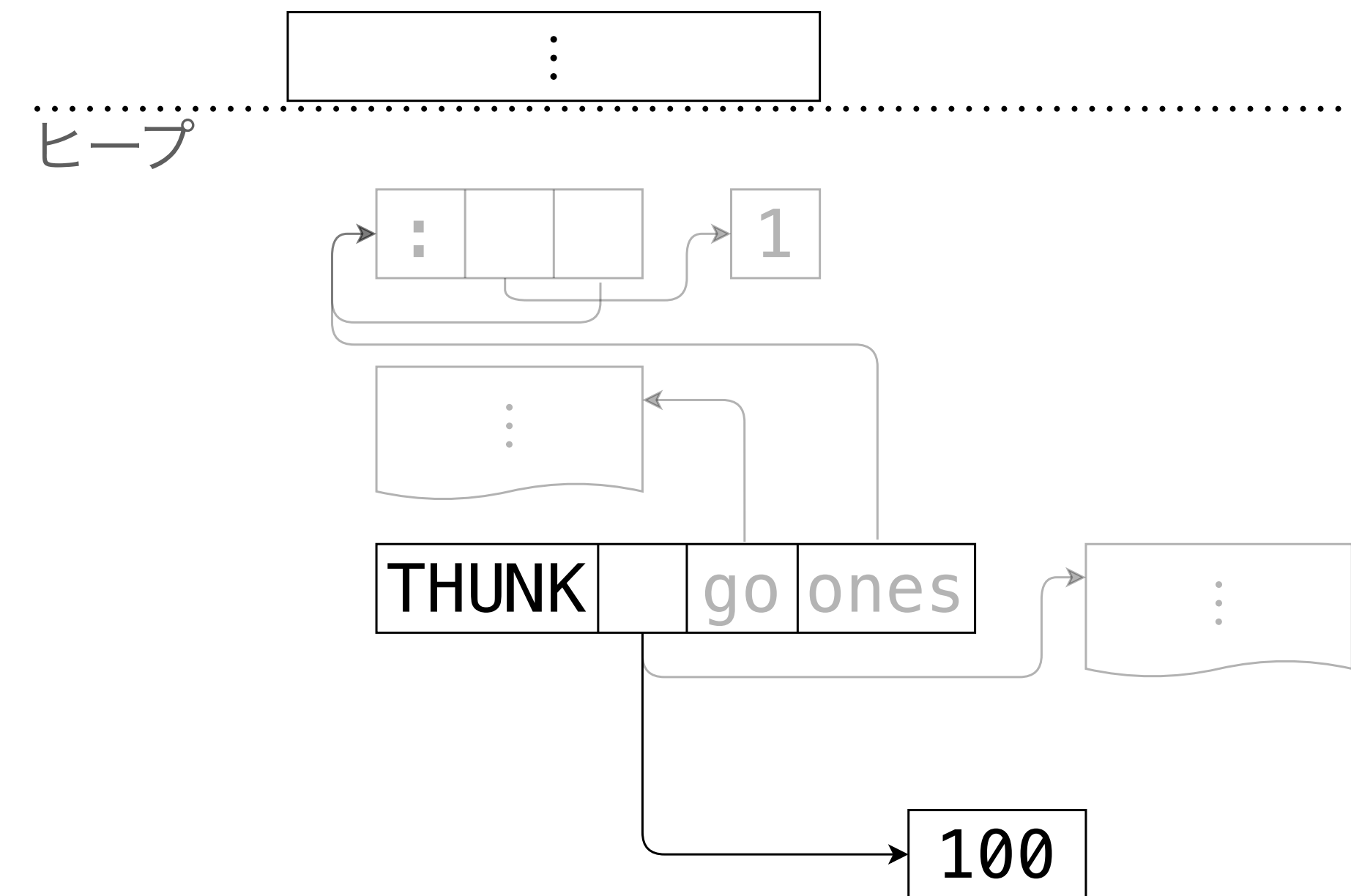
```
let ones = CON( (:) 1 ones) in
let go = FUN(z0 xs0 -> case (>=) z0 100 of {
  True -> z0;
  False -> case xs0 of {
    [] -> z0;
    (:) x xs ->
      let a1 = THUNK((+) x z0) in
      go a1 xs;
  });
}) in
let go_0_ones = THUNK(go 0 ones) in
print go_0_ones
```



# STG プログラムの動作

スタック

```
let ones = CON( (:) 1 ones) in
let go = FUN(z0 xs0 -> case (>=) z0 100 of {
    True -> z0;
    False -> case xs0 of {
        [] -> z0;
        (:) x xs ->
            let a1 = THUNK((+) x z0) in
            go a1 xs;
    });
}) in
let go_0_ones = THUNK(go 0 ones) in
print go_0_ones
```





```
putStrLn "Thank You for Listening!"
```

# 参考文献

- Simon Marlow. (2010). Haskell 2010 Language Report. Retrieved from <https://www.haskell.org/onlinereport/haskell2010/>
- Jones, S. L. P. (1992). Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(02), 127–202. <https://doi.org/10.1017/S0956796800000319>
- Marlow, S., & Jones, S. P. (2004). Making a fast curry: push/enter vs. eval/apply for higher-order languages. *ACM SIGPLAN Notices*, 39(9), 4. <https://doi.org/10.1145/1016848.1016856>
- Marlow, S., Yakushev, A. R., & Jones, S. P. (2007). Faster laziness using dynamic pointer tagging. *ACM SIGPLAN Notices*, 42(9), 277. <https://doi.org/10.1145/1291220.1291194>
- Maurer, L., Downen, P., Ariola, Z. M., & Peyton Jones, S. (2017). Compiling without continuations. *ACM SIGPLAN Notices*, 52(6), 482–494. <https://doi.org/10.1145/3140587.3062380>
- Takenobu, T. (2021). GHC (STG, Cmm, asm) Illustrated Rev. 0.05.0. Retrieved from [https://takenobu-hs.github.io/downloads/haskell\\_ghc\\_illustrated.pdf](https://takenobu-hs.github.io/downloads/haskell_ghc_illustrated.pdf)
- Generated Code. GHC Commentary. Retrieved from <https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/compiler/generated-code>
- STG Syntax. GHC Source Code. Retrieved from <https://gitlab.haskell.org/ghc/ghc/-/blob/ghc-9.0.1-release/compiler/GHC/Stg/Syntax.hs>