CS 520
Theory of Programming Language

06/02 – 06/09, 2021

Continuation in a Functional Language

1. Reminder / Overview.

① Continuation semantics.

$$[\![-]\!]_c \in [\langle exp\rangle \to E \to_c V_{cont} \to_c V_\star]$$

$$V_{cont} = V \to_c V_\star$$

$$V \cong V_{int} + V_{bool} + V_{fun} + V_{tuple} + V_{alt} + V_{cont}$$

$$V_{fun} = V \to_c V_{cont} \to_c V_\star$$

$[\![e]\!]_c \eta\, k$ = comp. of e together with the rest of comp. denoted by k.

$[\![3]\!]_c \eta\, k = k(\underbrace{\bar{\iota}_{int}(3)}_{V})$

.....

② Callcc, throw .... control operators.

③ Defunctionalization (first-order denotational semantics).

2. Callcc / throw

Label in C, Java ⋮ goto

{ function that takes a continuation

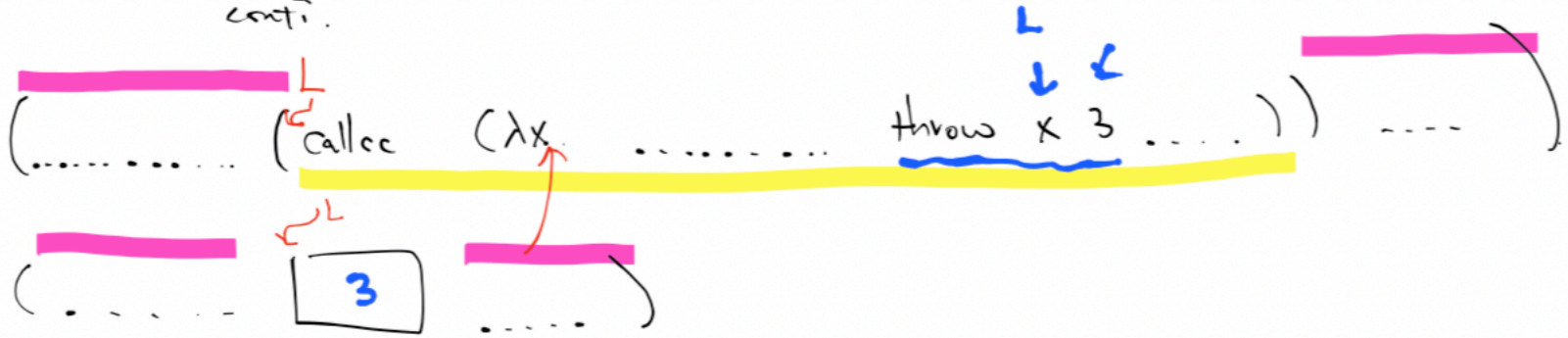
① <exp> ::= ... | callcc <exp> | throw <exp> <exp>

↑ continuation

② (callcc (λx.e)) , (throw e_1 e_2)

↑ current conti.

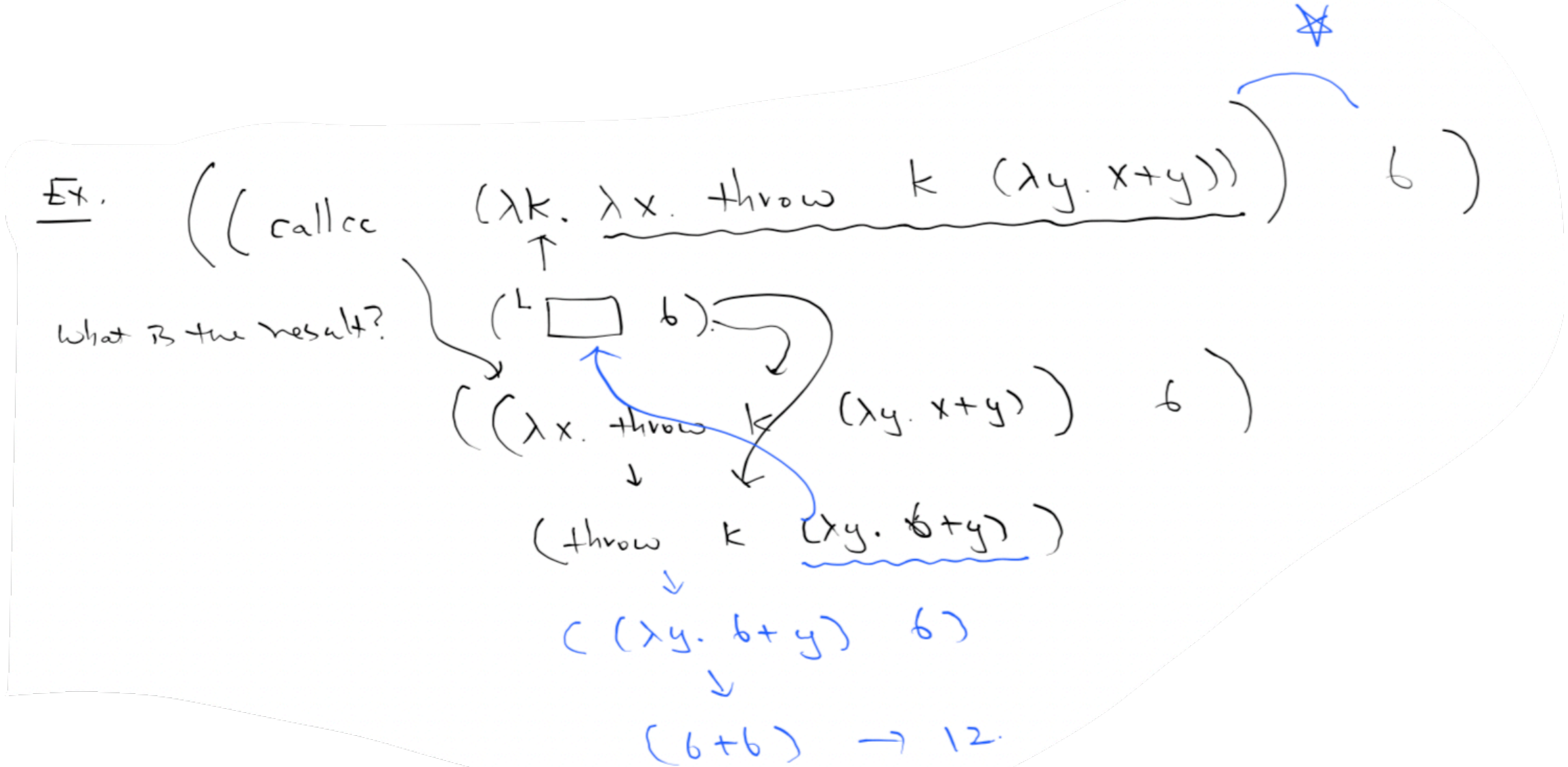
↑ k



(callcc (λk. 2 + (throw k (3 x 4))) + 20)

↓

(12 + 20) ⟶ 32


Ex.
((callcc (λk. λx. throw k (λy. x+y))) 6)
What is the result?
(□ 6)
((λx. throw k (λy. x+y)) 6)
(throw k (λy. 6+y))
((λy. 6+y) 6)
(6+6) → 12.

(3) Semantics.

(1) Change in the Bo. of V (+ V_{cont} case). $V_{fun} = [V \to_c V_{cont} \to_c V_*]$

(2) $[\![\text{callcc } e]\!]_c \eta\, k = [\![e]\!]_c \eta\, (\lambda f \in V_{fun}.\ f\ (\underbrace{\bar{v}_{cont}(k)}_{\text{ignored}})\ k\ )_{fun}$

$$\left(\begin{array}{l} \bar{v}_{cont} : V_{cont} \to V \\ \bar{v}_{cont}(k) = \psi \langle s, k \rangle \end{array} \right)$$

$[\![\text{throw } e_1\ e_2]\!]_c \eta\, k = [\![e_1]\!]_c \eta\, (\lambda k_1 \in V_{cont}.\ [\![e_2]\!]_c \eta\, k_1\ )_{cont}$

$\vdots$

k'

④ Extended example .... Backtracking.

tuple with 0 components.

true.

```
(backtrack (λamb. if (amb <>) then (if (amb <>) then 0
                                    else 1)
                  else if (amb <>) then 2 else 3))
```

false.

= [3 ; 2 ; 1 ; 0]

3 :: (2 :: (1 :: (0 :: nil)))

(a) 1 ⟨3,
  (a) 1 ⟨2,
    (a) 1 ⟨1,
      (a) 1 ⟨0, (a) 0 ⟨⟩ ⟩ ...⟩

(v fresh)

(1)

nil $\overset{\text{def}}{=}$ (a) 0 ⟨⟩

e :: e' $\overset{\text{def}}{=}$ (a) 1 ⟨e, e'⟩

(a) 0 ... nil or ::r ... (a) 1 ...

↓

listcase e of (e0, e1) ← (e1 c) r.

$\overset{\text{def}}{=}$ sumcase e of (λv. e0,
                    λv. (e1 v.0) v.1)

let x ≡ e in e' $\overset{\text{def}}{=}$ ((λx. e') e)

e ; e' $\overset{\text{def}}{=}$ let x=e in e' (for fresh x)

(2) <exp> ::= mkref <exp> | val <exp> | <exp> := <exp>

mkref 3 → a ↖ [3]

3 ↙ a

address a. value 10.

a [3] ⟿ a [10]

| <exp> =nf <exp>

a =nf b . . .

(3) backtrack $\stackrel{\text{def}}{=}$ λf. let rl ≡ mkref nil in. ← results of calling f under non-det.

let → cl ≡ mkref nil in.

list of continuations.

rl := f (λu. [dummy]) :: val rl ;

listcase (val cl) of amb

(val rl ,

λk. λl. cl := l ;

throw k false)

(callcc (λk. cl := k :: val cl ;

true)).

3. Semantic defunctionalization. — compiler. OO semantics.
(first-order denot. semantics).

$$V \cong \ldots + V_{fun} + V_{cont}.$$

$$V_{fun} \stackrel{def}{=} V \to_c V_{cont} \to_c V_*$$

$$[\![-]\!] \in [\langle exp \rangle \to E \to_c V_{cont} \to_c V_*]$$

Can we avoid this?

① Two key ideas. (1) Build spaces using syntax/instructions.
(2) Provide interpretations of these semantic instructions in those spaces.

② Overview for the lang with integer & fn canonical forms.

△ continuation.

$$\hat{V}_* = (\hat{V} + \{error, typeerror\})_\perp$$

$$\hat{V} \underset{\psi}{\overset{\phi}{\rightleftarrows}} V_{int} + \hat{V}_{fun} + \hat{V}_{cont}$$

$$V_{int} = \mathbb{Z}$$

↙ environments.

$$\hat{V}_{fun} \overset{def.}{=} \{abstract\} \times \langle var \rangle \times \langle exp \rangle \times \hat{E}$$

$$\Downarrow$$

$$\langle abstract, x, e, \eta \rangle \xrightarrow{*} \cdots$$

$$[\![-]\!]$$

$$* \; apply : \hat{V}_{fun} \to_c [\hat{V} \to_c \hat{V}_{cont} \to_c \hat{V}_*]$$