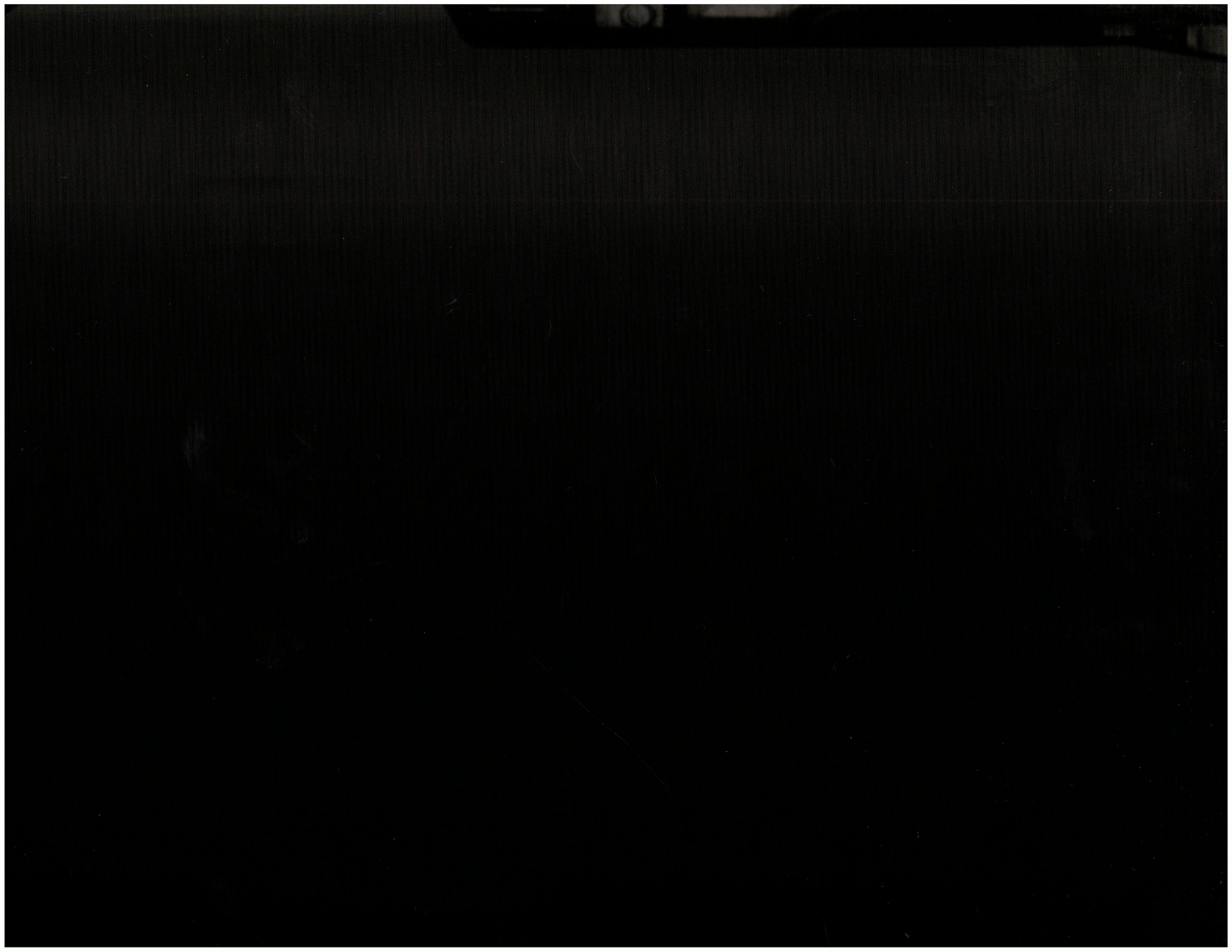


Intermediate Compiler Forms for Functional Languages

Jarbd Gentner

1. What is an intermediate representation?
2. Why is Continuation Passing Style (CPS) such an effective one?
3. How can we get the benefits of CPS without the drawbacks?
4. Why do compiler writers love A-Normal Form (ANF)?
5. Can we do everything in ANF that we could in CPS?

Compiler Pipeline

Parse : String → AST

Desugar : AST → AST

Typecheck : AST → AST

[

Simplify : AST → L

Optimize : L → L

Generate : L → Assembly

]

Intermediate Representation

QUAD CODES

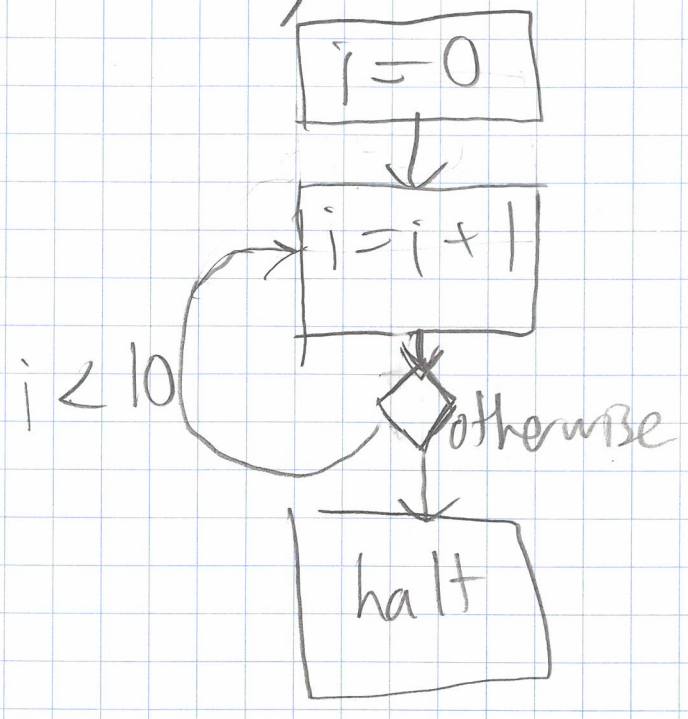
Simplify: AST → QUAD
Optimize: QUAD → QUAD
Generate: QUAD → Assembly

1) i = 0
2) i = i + 1
3) if i ≤ 10 goto 2
4) halt

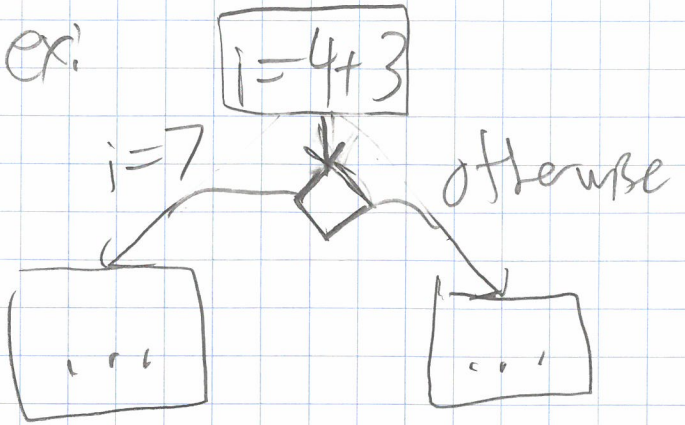
1) i = 4 + 3
2) if i = 7 goto n
...
n-1) halt
n)
...
m) halt

CONTROL FLOW GRAPH

Simplify: AST → CFG
Optimize: CFG → CFG
Generate: CFG → Assembly



ex:



$$e ::= v \mid e\ e \mid \text{if } e \text{ then } e \text{ else } e \mid (\text{prim } e \ldots) \mid \text{let } x = e \text{ in } e$$

$$v ::= x \mid \lambda x.e \mid c$$

$$(\lambda x.e_1)\, e_2 \longrightarrow e_1[e_2/x]\ \beta$$

or

$$(\lambda x.e_1)\, v \longrightarrow e_1[v/x]\ \beta_v$$

(let, primitives, if ...)

What is an optimization?

$e_1 \longrightarrow e_2$

$e_1 \cong e_2$

e_2 is "faster" than e_1

$(\lambda x.\, x)\,(print\ 42) \longrightarrow_\beta (prnt\ 42)$

not possible w/ β_v

$(\lambda x.\, 0)\,(prnt\ 42) \longrightarrow_\beta 0$

unsound in almost all langs...

CONTINUATION PASSING STYLE

Van Wijngaarden ['64]

"Provide each procedure declaration with an extra formal parameter – specified **label** – and insert at the end of its body a goto statement leading to that formal parameter."

Fischer ['72]

Reynolds ['72]

Plotkin ['77]

$e_1 \xrightarrow{\beta v}{}^{*} e_2 \Rightarrow CPS[e_1] \xrightarrow{\beta n}{}^{*} CPS[e_2]$,

but not the converse.

CPS COMPILER HISTORY

RABBIT [STEELE '78]

- Pioneering the technique
- Toy

ORBIT [YAIR '87]

- Production compiler
- Faster than CL!

SML/NJ [MacQueen et al. '90]

- Based on Orbit
- Written up in Compiling w/ Continuations

let $f\ x = e_2$ in $e_1 \rightarrow$ let $f = \lambda x.\ e_2$ in e_1

let $f\ x\ k = \ldots$

k: continuation parameter

$CPS_k[\lambda x.\ e] =$

  let $f\ x\ j = CPS_j[e]$ in $k\ f$

$CPS_k[f\ x] =$

  let $j\ f' =$

    let $l\ x' = f'\ x'\ k$ in $CPS_l[x]$

  in $CPS_j[f]$

$CPS_k[v] = k\ v$

$CPS[(\lambda x.x)\ print\ 42] =$

$let\ j\ f' =$
  $let\ l\ x' = f'\ x'\ k\ in$
    $print_k\ 42\ l$
$in\ let\ f\ x\ m = m\ x\ in\ j\ f$

$\xrightarrow{\beta}$

$let\ l\ x = k\ x\ in\ print_k\ 42\ l$

$\xrightarrow{\eta}$

$print_k\ 42\ k$

$CPS[(\lambda x.0)\ print\ 42] =$

$let\ l\ x = k\ 0\ in\ print_k\ 42\ l$

$CPS[\![(+\ (+\ 1\ 2)\ (+\ 3\ 4))]\!] =$

$(+_k\ 1\ 2\ (\lambda l.$
$(+_k\ 3\ 4\ (\lambda r.$
$(+_k\ l\ r\ (\lambda(x)\ x))))$

QUAD:

$l \leftarrow 1 + 2$
$r \leftarrow 3 + 4$
$x \leftarrow l + r$
halt.

$*+_k(l, r, k) = k\ (+\ l\ r)$

Good for code generation!

COMPILING WITH CONTINUATIONS [APPEL '92]

2 goals:

Argue from theory that CPS is best IR
- Plotkin shows issues w/ λ_v
- CPS closed under β reduction
- Many optimizations are β in CPS
  - Constant folding
  - Common subexpression elimination
  - Inlining

Advocate for a particular way of CPSing
- Continuations are just functions
- so-called "stackless" strategy
- code is simpler, but at what cost?

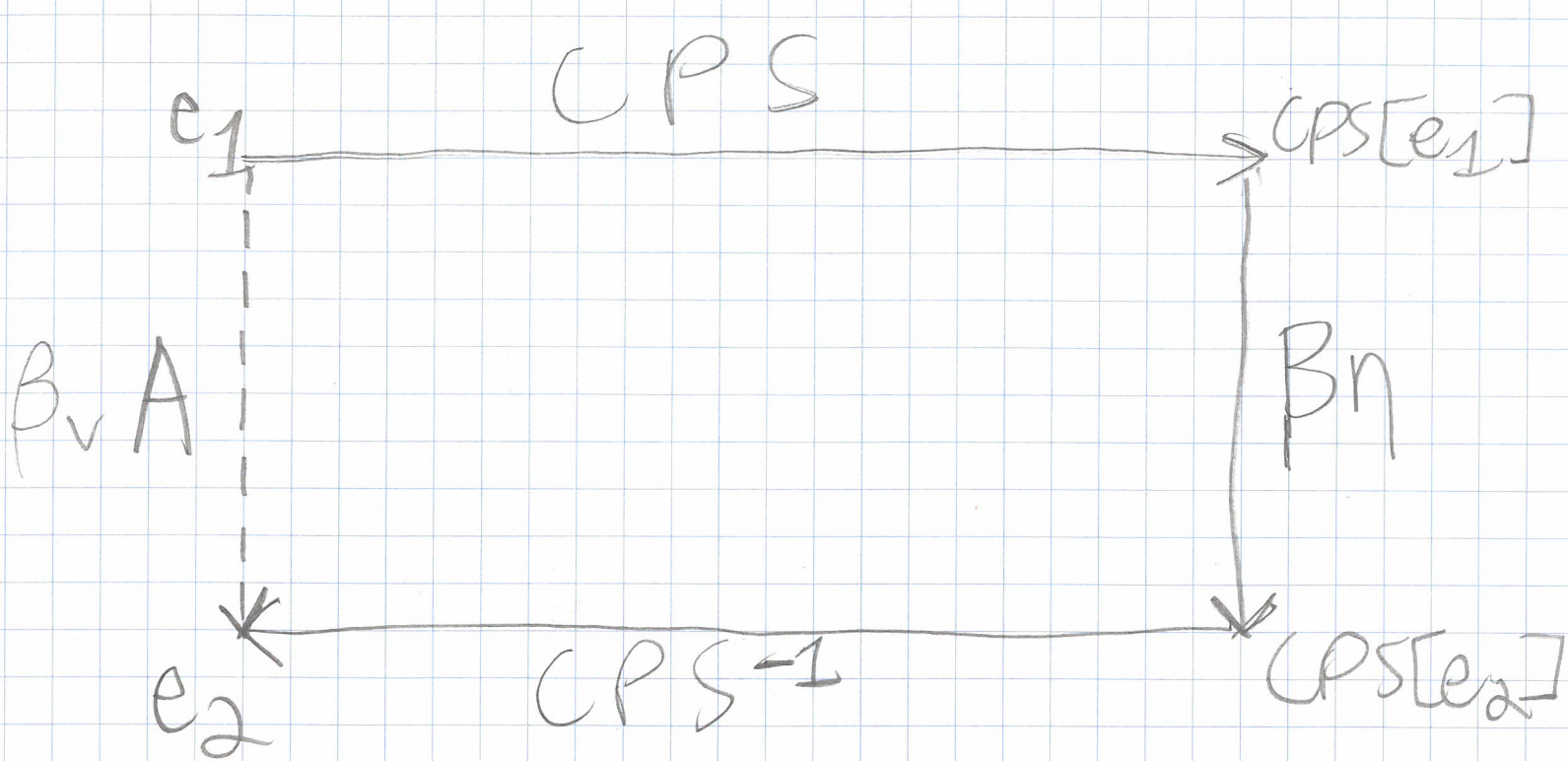
REASONING ABOUT PROGRAMS IN CPS
SABRY & FELLEISEN ['92]

Recall:

$$CPS[e_1] \xrightarrow[\beta,\eta]{}{}^{*} CPS[e_2] \not\Rightarrow e_1 \xrightarrow[\beta_v]{}{}^{*} e_2$$

Can we find some axioms A, such that:

$$CPS[e_1] \xrightarrow[\beta,\eta]{}{}^{*} CPS[e_2] \Rightarrow e_1 \xrightarrow[\beta_v, A]{}{}^{*} e_2$$



Questions:

- What do administrative redexes look like in direct style?
- How do we get rid of them without $\beta\eta$?

A bit of context...

(+ 10 (- (* 6 6) 4))

(- (* 6 6) 4)

(* 6 6) ⇒ 36

(+ 10 (- 36 4))

(- 36 4) ⇒ 32

(+ 10 32) ⇒ 42

E = □

E = (+ 10 □)

E = (+ 10 (- □ 4))

E = □

E = (+ 10 □)

E = □

```
let m x = (+ 10 x k) in
  let n x = (- k x 4 m) in
    (* k 6 6 n)
```

Name each subcomputation

(+ 10 (- (* 6 6) 4)) ⟹

```
let x1 = (* 6 6) in
let x2 = (- 4 x1) in
let x3 = (+ 10 x2) in
x3
```

$E[((f\ a)\ b)] \longrightarrow$ let x = f a in E[x b]

β-flat

Just one more thing...

$$f\ (\texttt{let}\ x = 42\ \texttt{in}\ x) \longrightarrow$$

$$\texttt{let}\ j\ v = f\ v\ k\ \texttt{in}\ \texttt{let}\ x = 42\ \texttt{in}\ j\ x$$

$$E[\texttt{let}\ x = e\ \texttt{in}\ b] \xrightarrow[\beta\text{-lift}]{} \texttt{let}\ x = e\ \texttt{in}\ E[b]$$

$\beta_{\text{left}}\beta_{\text{flat}}$ normal form performs admn. reductions

What about $\beta\eta$?

Apply CPS^{-1} to β/η rules

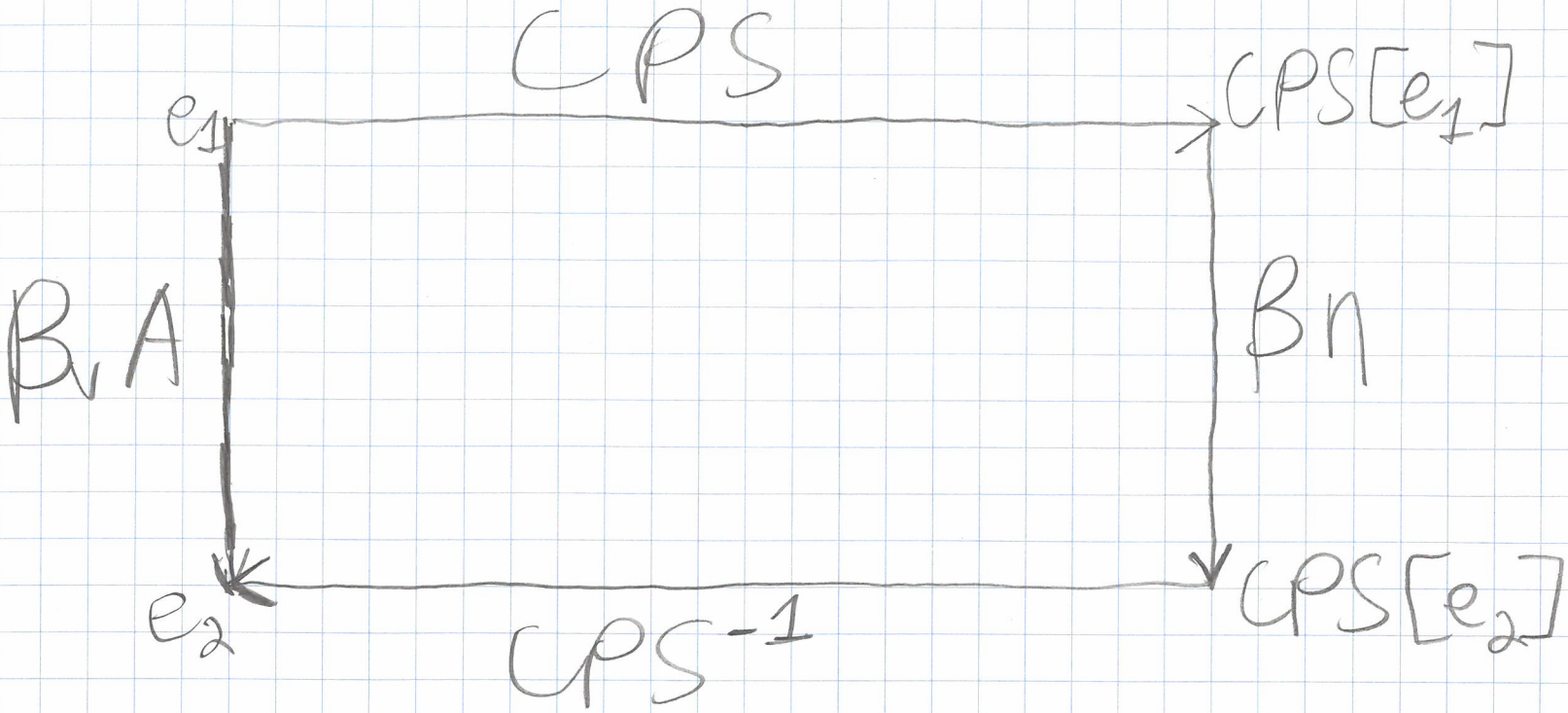
Yields: $((\lambda x.x)\ e) \to e$ β_{id}

$\text{let } x = e \text{ in } E[f\ x] \xrightarrow{} E[f\ e]$ β_Ω *

$(\lambda x.x)(\text{print } 42) \xrightarrow{\beta_{id}}$

print 42

* x must only occur once in the body and f is a name



$$A = \{\eta_v, \beta_{lift}, \beta_{flat}, \beta_{id}, \beta_\Omega\}$$

THEOREM: A reductions are strongly normalizing

THE ESSENCE OF COMPILING WITH CONTINUATIONS

[FLANAGAN et. al '93]

- Direct style compilers use elements of ANF
- CPSing compilers undo CPS transform

```
let m x = (+ 10 x k) in
let n x = (- x 4 m) in
(* 6 6 n)
```

} Heap allocated

```
let x1 = (* 6 6) in
let x2 = (- 4 x1) in
(+ 10 x2)
```

} Stack allocated

- Provides linear-time ANFing algorithm

```
ANF[λx. e] k =
  let f x = ANF[e] id in k f
```

```
ANF[f x] k =
  ANF[f] (λf'.
    ANF[x] (λx'.
      let n = f' x'
      in k n))
```

```
ANF[v] k = k v
```

ANF IN REVIEW

- Extends λ_v to have same reasoning power as λ_{CPS}
- "Think in CPS, work in direct-style"
- Preserves stack, making naïve ANF faster
  - CPS is largely abandoned.

    Compilers that used or use ANF:
    - MLTon (90s)
    - OCaml (Now)
    - Haskell (Now)
    - Pyret

REVIVAL OF CPS

- Contification using Dominators [Fluet + Weeks 2001]
  - ANF inside first order continuations
    - * Why? Named contexts.

If a function calls the same local continuation at all exits, the cont. can be compiled to a jump

- Tail call elimination (but more general)

COMPILING WITH CONTINUATIONS, CONTINUED [KENNEDY '07]

- Shows a (relatively) simple way to compile CPS efficiently
  - Second class conts., stack allocated, or jumps
- Two arguments against ANF:
  - * β reduction requires re-normalization
    - Can have $O(n^2)$ runtime
  - * If statements pose a problem for ANF

"Though curiously, the 'A-normalization algorithm' in Flanagan et. al does not actually normalize terms, as it leaves let bound conditionals alone."

```
ANF[if c then t else e] k =
  ANF[c] (λc'.
    k (if c' then ANF[t] id else ANF[e] id))
```

```
ANF[...] k =
  ANF[c] (λc'.
    (if c' then ANF[t] k else ANF[e] k))
```

duplicated

```
ANF[...] k =
  ANF[c] (λc'.
    let j x = k x in
    if c' then ANF[t] (λn. j n) else ANF[e] (λn. j n))
```

continuation

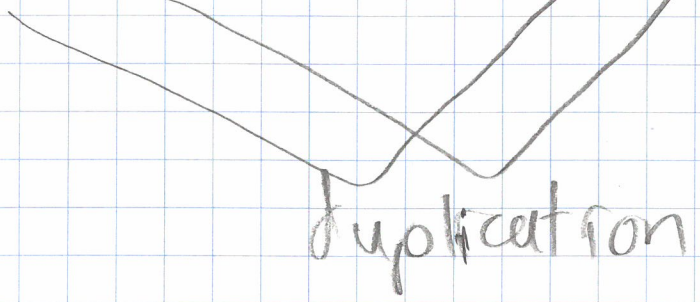
Commuting Conversion:

(if (if a b c) d e) ⟶
(if a (if b d e) (if c d e)) cc

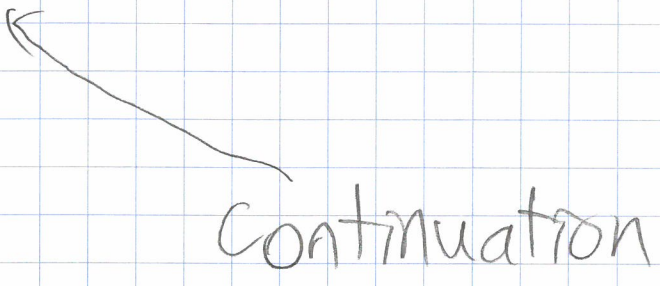
ex: not ≡ λx. if x #f #t

(if (not x) 0 42)
⟶ not
(if (if x #f #t) 0 42)
⟶ cc
(if x (if #f 0 42) (if #t 0 42))
⟶ if
(if x 42 0)

(if a (if b d e) (if c d e))




```
let k x = if x then d else e
in
  if a then k d else k e
```



CPS_k[if c then t else e] =

let j c' = if a' then CPS_k[t] else CPS_k[e]
in CPS_j[c]

CPS_k[if (if a then b else c) then d else e] =

let j x' = if x' then k d else k e in
let m a' = if a' then j b else j c in
m a

$\xrightarrow{\beta}$

let j x' = if x' then k d else k e in
if a then j b else j c

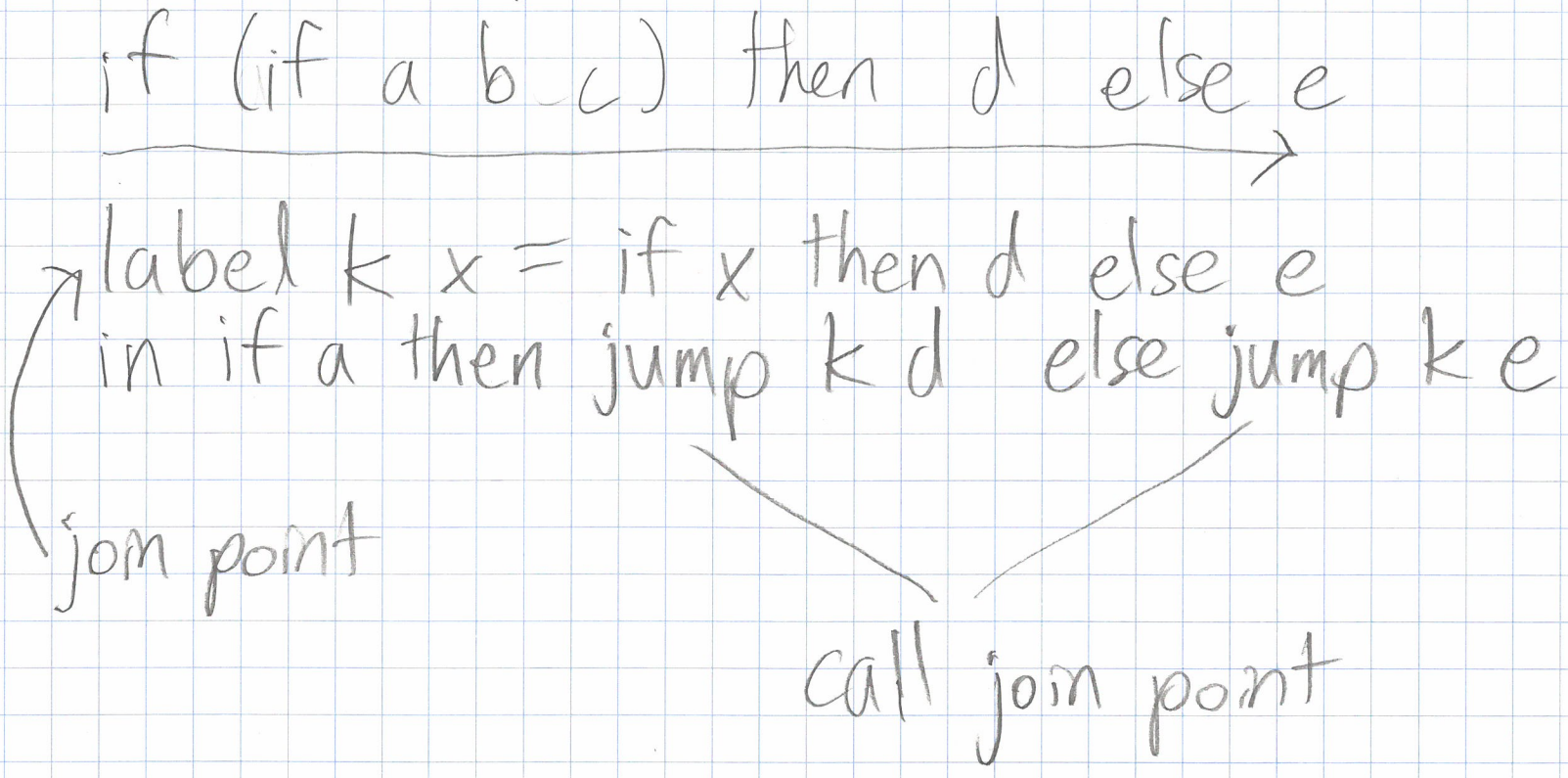
$\xrightarrow{\beta}$

if a then (if b then k d else k e)
(if c then k d else k e)

COMPILING WITHOUT CONTINUATIONS
[MAURER et al. '17]

What's missing? 1 small use of conts.

Define "join points" from compiler folklore



Expresses contification in terms of join points

```
if (label k x = if x then d else e
    in if a then jump k b else jump k c)
then 42 else 0
```

$\longrightarrow \beta_{lift}$

```
label k x = if x then d else e
in if (if a then jump k b else jump k c)
then 42 else 0
```

$\longrightarrow cc$

```
label k x = if x then d else e in
label j x = if x then 42 else 0 in
if a then jump j (jump k b)
     else jump j (jump k c)
```

CPS_k[if (if (if a then b else c)
then d
else e)
then 42
else 0]

→

let j x' = if x' then k 42 else k 0 in
let m x' = if x' then j d else j e in
let n a' = if a' then m b else m c in
n a

→ β

let m x' = if x' then
if d then k 42 else k 0
else
if e then k 42 else k 0 in
if a then m b else m c

```
if (label k x = if x then d else e
    in if a then jump k b else jump k c)
then 42 else 0
```

→ cc

```
label k x = if x then
              if d then 42 else 0
            else
              if e then 42 else 0
in if a then jump k b else jump k c
```

Think in CPS, work in direct style!

THE CONCLUSION

ANF vs. CPS: Anything you can do, I can do better.
(or at least the same).

Dialogue of THEORY... and PRACTICE
- Plotkin
- λvA
- Heap vs. stack
- jumps

Competitive Cooperation

Real Clients, Real Consequences