commuting conversions + join points in call-by-push-value

NJPLS @ Princeton, 5 December 2025

Jonathan Chan
Univ. of Pennsylvania

Madi Gudin
Amherst College

Annabel Levy
UMBC

Stephanie Weirich
Univ. of Pennsylvania

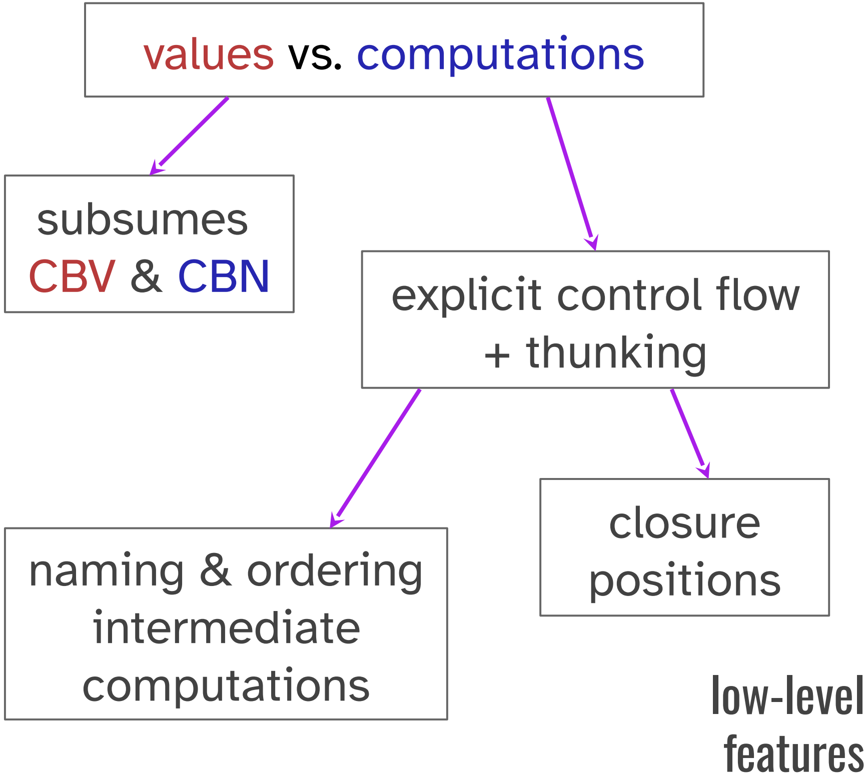
λREPL
penn-repl.github.io

the big picture:

CBPV as a compiler IR

† Paul B. Levy (2001, 2004)

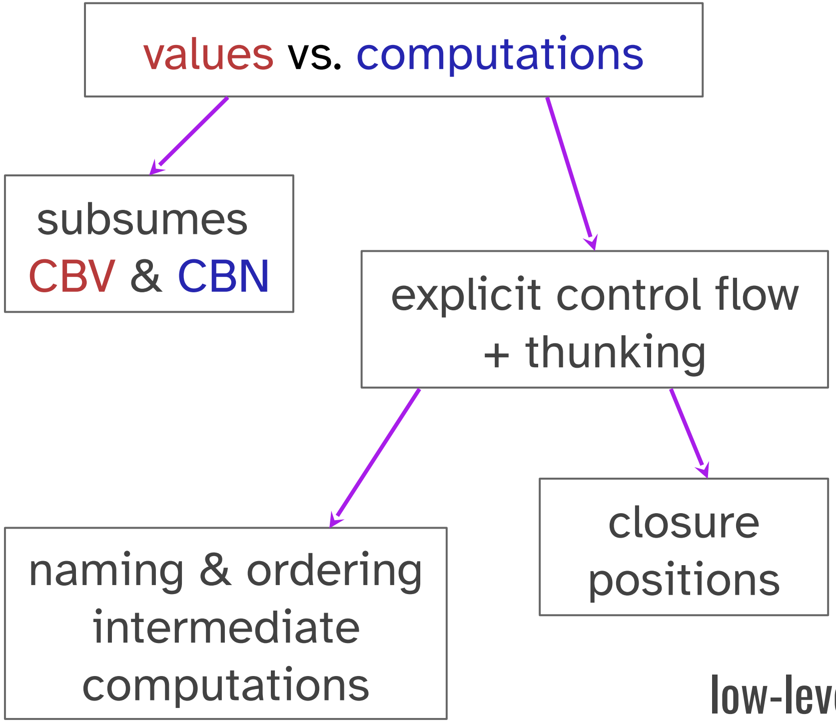
CBPV† as compiler IR



high-level features

† Paul B. Levy (2001, 2004)

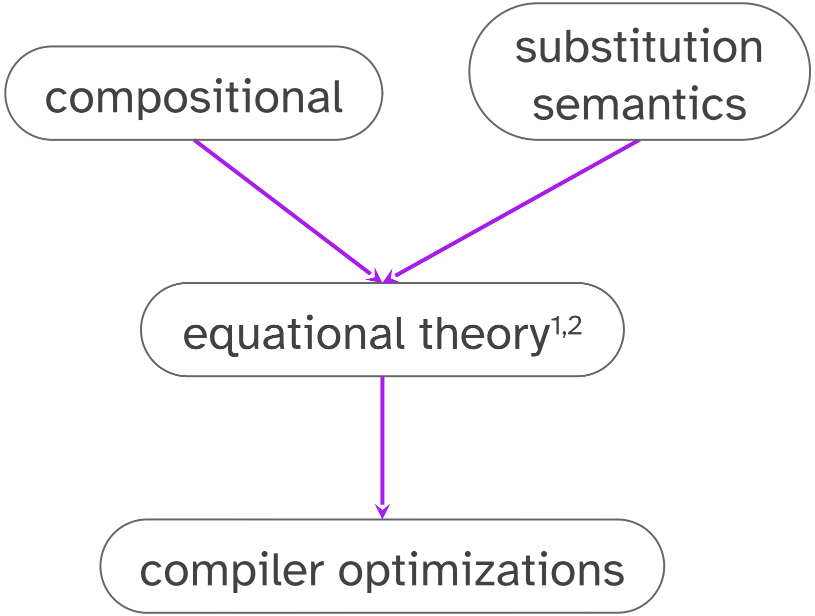
CBPV† as compiler IR



low-level features

high-level features

[1] Rizkallah, Garbuzov, Zdancewic (2018)
[2] Forster, Schäfer, Spies, Stark (2019)



```
(λx. x + 3) ((λy. y + 1) 1)
```

call by value translation

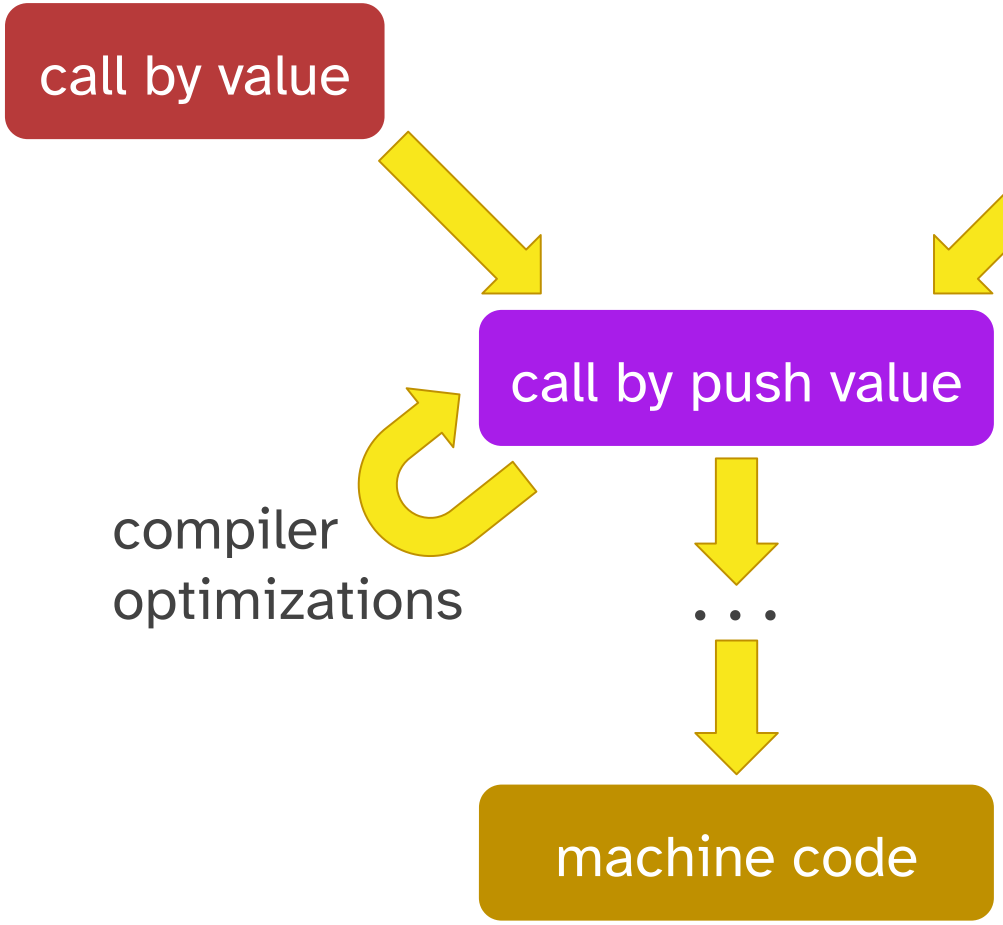
lambda calculus

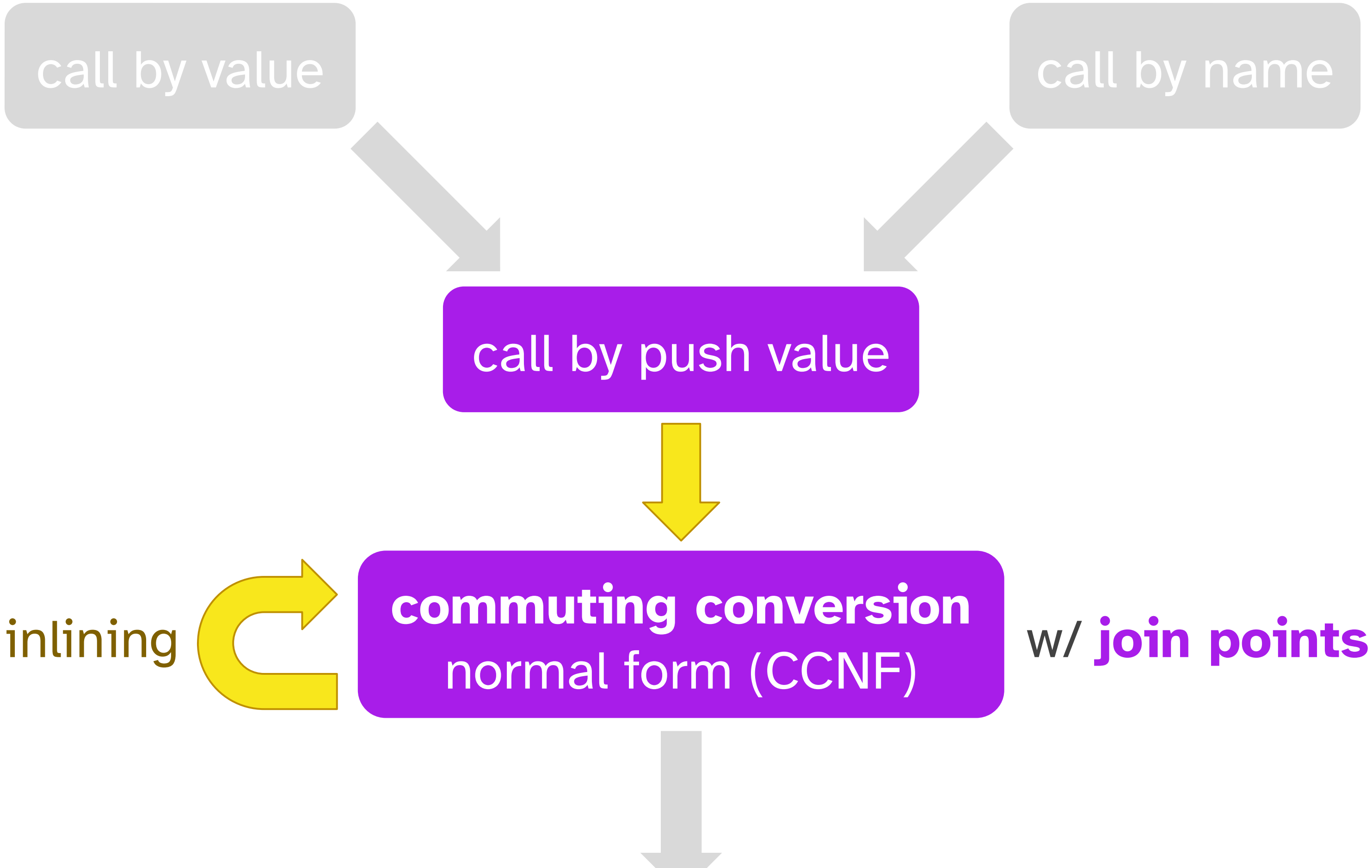
call-by-push-value

call by name translation

```
let z ← (λy. ret (y + 1)) 1
in (λx. ret (x + 3)) z
```

```
(λx. let x’ ← x!
     in ret (x’ + 3))
{(λy. let y’ ← y!
      in ret (y’ + 1))
 {ret 1}}
```


call by value
call by name
call by push value
compiler
optimizations
. . .
machine code


call by value
call by name
call by push value
inlining
commuting conversion
normal form (CCNF)
w/ join points

CPBV + commuting conversions + join points

- What **commuting conversions** are
- Why we need **join points**
- When *inlining* comes in (β-reductions)

❶ commuting conversions

```
app-let |   (let x ← n in    n’  ) v
        | ⇒ let x ← n in ((  n’  ) v)
```

push elimination forms inside tail positions

❶ commuting conversions

```
           (let x ← n in    n’ ) v
app-let  ⇒ let x ← n in ((  n’ ) v)

           let y ← (let x ← n in    n’ ) in m
let-let  ⇒ let x ← n in (let y ←    n’   in m)
```

push elimination forms inside tail positions

❶ commuting conversions + inlining

```
           (let x ← n in λy. m) v
app-let  ⇒ let x ← n in ((λy. m) v)
      β  ⇒ let x ← n in m[y ↦ v]

           let y ← (let x ← n in ret v) in m
let-let  ⇒ let x ← n in (let y ← ret v in m)
      β  ⇒ let x ← n in m[y ↦ v]
```

commuting conversions expose inlining opportunities

❷ commuting conversions

```
           let y ← (if v then n₁ else n₂) in m
let-if  ⇒ if v then (let y ← n₁ in m)
               else (let y ← n₂ in m)

           …
app-if  ⇒  …
```

❷ commuting conversions + join points

let-if

```
  let y ← (if v then n₁ else n₂) in m
⇒ if v then (let y ← n₁ in m)
            else (let y ← n₂ in m)
⇒ let f ← ret {λy. m} in
  if v then (let y ← n₁ in f! y)
        else (let y ← n₂ in f! y)
```

❷ commuting conversions + join points

let-if

```
   let y ← (if v then n_1 else n_2) in m
⇒ if v then (let y ← n_1 in m)
         else (let y ← n_2 in m)
⇒ let f ← ret {λy. m} in
   if v then (let y ← n_1 in f! y)
         else (let y ← n_2 in f! y)
```

incurs cost of creating closure
function calls may be expensive

❷ commuting conversions + primitive join points

let-if

```
   let y ← (if v then n₁ else n₂) in m
⇒ if v then (let y ← n₁ in m)
           else (let y ← n₂ in m)
⇒ join j y = m in
   if v then (let y ← n₁ in jump j y)
           else (let y ← n₂ in jump j y)
```

conditions:
no jumps inside thunks
jumps only in tail position

Compiling without Continuations, PLDI 2017

❸ commuting conversions in little steps

```
        | (let x ← n in (λw. (let y ← n in (λz. m)) w)) v
app-let | ⇒ let x ← n in ((λw. (let y ← n in (λz. m)) w) v)
      β | ⇒ let x ← n in ((let y ← n in (λz. m)) v)
app-let | ⇒ let x ← n in let y ← n in ((λz. m) v)
      β | ⇒ let x ← n in let y ← n in m[z ↦ v]
```

❸ commuting conversions all at once

```
           (let x ← n in (λw. (let y ← n in (λz. m)) w)) v
app-let* ⇒ let x ← n in ((λw. let y ← n in ((λz. m) w)) v)
       β ⇒ let x ← n in let y ← n in (λz. m) v
       β ⇒ let x ← n in let y ← n in m[z ↦ v]
```

single-pass commuting conversion normalization

to CCNF subset of CBPV + join points

$[\![\cdot]\!]K$	$[\![\cdot]\!]$	*CC-normalization*
$\Gamma \mid \Delta \vdash m : B$	$\Gamma \vdash v : A$	*typing*
$m \rightsquigarrow^* m'$		*evaluation*

single-pass commuting conversion normalization

to CCNF subset of CBPV

$[\![\cdot]\!]K$ $[\![\cdot]\!]$ *CC-normalization*

$\Gamma \mid \Delta \vdash m : B$ $\Gamma \vdash v : A$ *typing*

$m \rightsquigarrow^* m'$ *evaluation*

$\Gamma \mid \Delta \vDash m_1 \sim m_2 : B$ *semantic equivalence*

Fundamental lemma:

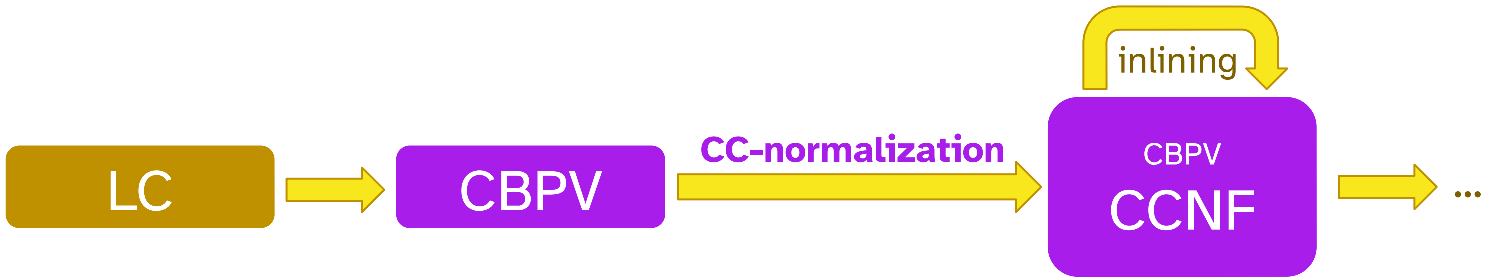
If $\Gamma \mid \cdot \vdash m : B$ then $\Gamma \mid \cdot \vDash m \sim [\![m]\!]\square : B$.

Equivalence of CC-normalization:

If $\cdot \mid \cdot \vdash m : F\ A$ then $m \rightsquigarrow^* \mathtt{ret}\ v\ {}^*\!\leftsquigarrow [\![m]\!]\square$.

metatheory

mechanized in Lean 4



commuting conversions + join points in CBPV

paper draft: https://ionathan.ch/assets/pdfs/ccnf.pdf
mechanization: https://github.com/ionathanch/CBPV/tree/join