Declarative Pearl:

Deriving Monadic Quicksort

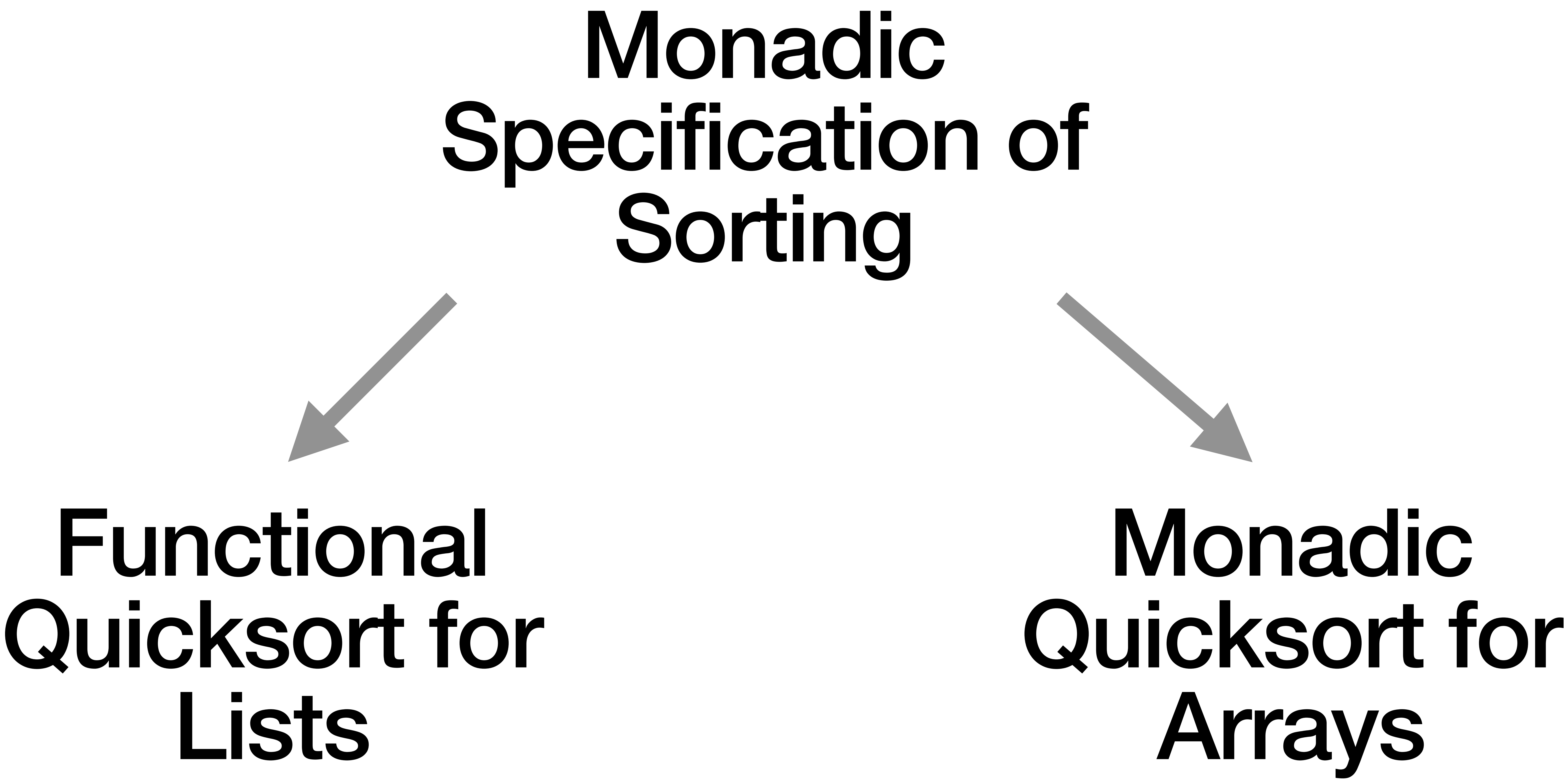
Shin-Cheng Mu and Tsung-Ju Chiang,
Academia Sinica, Taiwan.

FLOPS 2020.

Monadic Specification of Sorting

Monadic Specification of Sorting

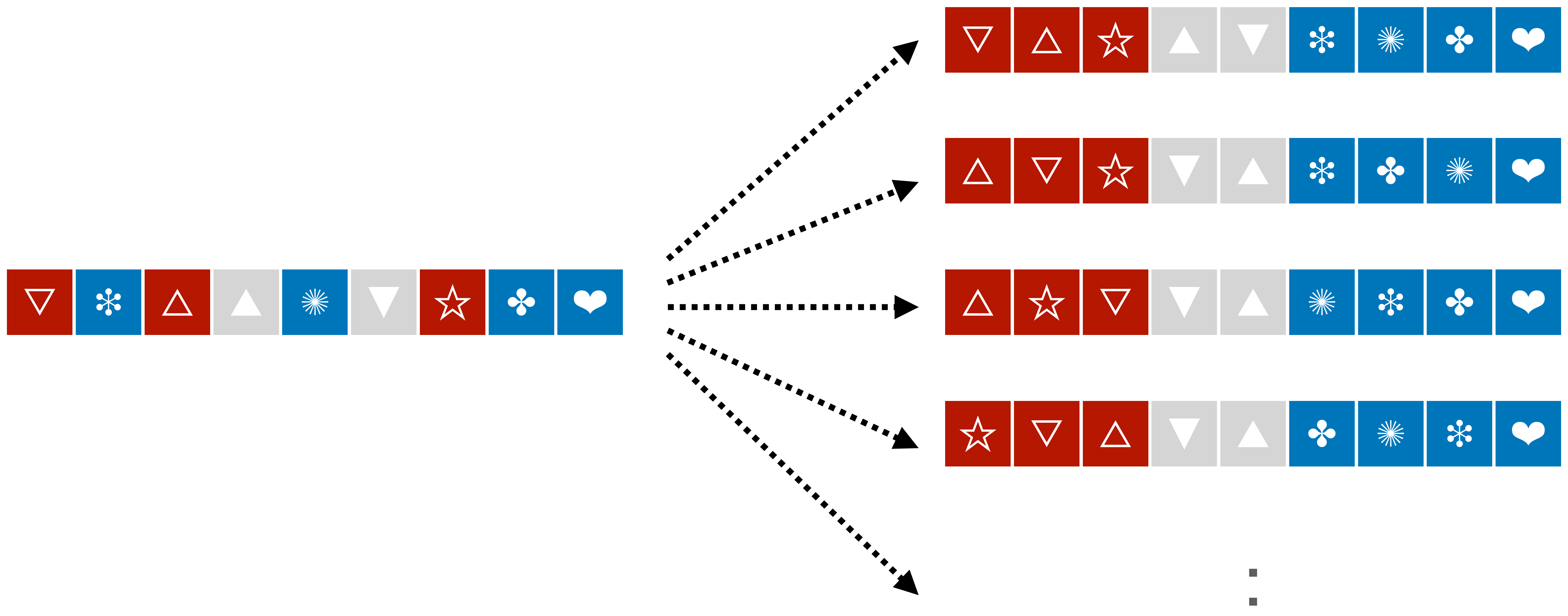
Functional Quicksort for Lists


Monadic Specification of Sorting
Functional Quicksort for Lists
Monadic Quicksort for Arrays

Program Derivation

$$
\begin{array}{ll}
 & \text{problem specification} \\
= & \quad \{ \text{ reason 1 } \} \\
 & \text{expr 1} \\
= & \quad \{ \text{ reason 2 } \} \\
 & \text{expr 2} \\
 & \quad \vdots \\
= & \quad \{ \text{ reason n } \} \\
 & \text{implementation.}
\end{array}
$$

Non-determinism



Some sorting algorithms can be more efficient if we are not over specific about order of items with equal keys.

Relational Program Derivation

relational specification

$\supseteq$ { reason 1 }

expr 1

$=$ { reason 2 }

expr 2

⋮

$\supseteq$ { reason n }

functional implementation.

$R \supseteq S$: relational inclusion;
whatever S does is allowed by R.

Relational Program Derivation

Developed around late 80's - 90's.

I still love it!

"Bizarre, too complex."

Point-free. Harder to apply usual techniques such as pattern matching, induction on the arguments, etc.

Pointwise notation confusing when applying a function to a non-deterministic value.

Monadic Program Derivation

$$
\begin{array}{ll}
 & \textbf{Monadic-Spec}\ (x{:}xs) \\
\supseteq & \quad \{\ \text{reason } 1\ \} \\
 & \text{expr } 1 \\
= & \quad \{\ \text{reason } 2\ \} \\
 & \text{expr } 2 \\
 & \quad \vdots \\
\supseteq & \quad \{\ \text{reason } n\ \} \\
 & \textbf{return}\ (\text{implementation}\ (x{:}xs))\ .
\end{array}
$$

$R \supseteq S$: to be defined later!

Monadic Program Derivation

Non-determinism represented by monads.

One can apply usual functional program derivation techniques --- e.g. structural induction on input data.

Possibility of incorporating other effects.

Monad

operators

$$\text{return} :: a \rightarrow m\ a$$
$$(\gg\!=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$$

monad laws

$$\text{return}\ x \gg\!= f \quad = \quad f\ x$$
$$f \gg\!= \text{return} \ = \ f$$
$$(m \gg\!= f) \gg\!= g \ = m \gg\!= (\lambda\ x \rightarrow f\ x \gg\!= g)$$

Non-determinism Monad

operators

(▯) :: m a → m a → m a

∅ :: m a

Non-determinism Monad

operators

$$(\square) \;::\; m\ a \to m\ a \to m\ a$$

$$\varnothing \;::\; m\ a$$

monoid

$$(m \;\square\; n) \;\square\; k \;=\; m \;\square\; (n \;\square\; k)$$

$$m \;\square\; \varnothing = m = \varnothing \;\square\; m$$

idempotence

$$m \;\square\; m = m$$

commutativity

$$m \;\square\; n = n \;\square\; m$$

Insertion & Permutation

```
insert :: a → [a] → m [a]
insert y [ ]      = return [y]
insert y (x:xs) = return (y:x:xs) ▯
           (x:) ⟨$⟩ insert y xs
```

Insertion & Permutation

```
insert :: a → [a] → m [a]
insert y [ ]      = return [y]
insert y (x:xs) = return (y:x:xs) ▯
            (x:) ⟨$⟩ insert y xs

f ⟨$⟩ m  =  m ≫= (λ x → return (f x))
```

Insertion & Permutation

```
insert :: a → [a] → m [a]
insert y [ ]      = return [y]
insert y (x:xs) = return (y:x:xs) ▯
          (x:) ⟨$⟩ insert y xs

f ⟨$⟩ m  =  m ≫= (λ x → return (f x))

perm :: [a] → m [a]
perm [ ]      = return [ ]
perm (x:xs) = perm xs ≫=  insert x
```

Sorting

$$\text{slowsort } xs = \text{perm } xs \gg\!= \text{filt ordered}$$

$$\begin{aligned}\text{filt } p\ x &= \textbf{if } p\ x \textbf{ then } \text{return } x \textbf{ else } \varnothing \\ &= \text{guard } (p\ x) >> \text{return } x\end{aligned}$$

$$\text{guard } b = \textbf{if } b \textbf{ then } \text{return } (\,) \textbf{ else } \varnothing$$

Program Refinement

$$m_1 \subseteq m_2 \;\equiv\; (m_1 \,\square\, m_2 = m_2)$$

Divide & Conquer

```
perm [ ] = {[ ]}
perm (x:xs) =
      split xs ≫= λ (ys,zs) →
      perm ys ≫= λ ys' →
      perm zs ≫= λ zs'→
      liftM2 (⧺[x]⧺) ys' zs'
```

splits : all the ways to split a list into two.

Deriving Quicksort

$$\begin{aligned}
\text{slowsort } [\,] &= \{[\,]\} \\
\text{slowsort } (x{:}xs) &= \text{perm } (x{:}xs) \gg\!= \text{filt ordered} \\
&= \,?
\end{aligned}$$

```
slowsort (x:xs) =
    perm (x:xs) >>= filt ordered
```

```
slowsort (x:xs) =
    split xs >>= λ (ys,zs) →
    perm ys >>= λ ys' →
    perm zs >>= λ zs' →
    filt ordered (ys' ++ [x] ++ zs')
```

```
slowsort (x:xs) =
    split xs >>= λ (ys,zs) →
    perm ys >>= λ ys' →
    perm zs >>= λ zs' →
    guard (ordered ys' ∧ ordered zs' ∧
           all (≤x) ys' ∧ all (≥x) zs')) >>
    return (ys' ++ [x] ++ zs')
```

```
slowsort (x:xs) =
    split xs >>= λ (ys,zs) →
    guard (all (≤x) ys' ∧ all (≥x) zs')) >>
    (perm ys >>= filt sorted)>>= λ ys' →
    (perm zs >>= filt sorted)>>= λ zs'→
    return (ys' ++ [x] ++ zs')
```

```
slowsort (x:xs) =
    split xs >>= λ (ys,zs) →
    guard (all (≤x) ys' ∧ all (≥x) zs')) >>
    slowsort ys >>= λ ys' →
    slowsort zs >>= λ zs'→
    return (ys' ++ [x] ++ zs')
```

```
slowsort (x:xs) =
  split xs >>= λ (ys,zs) →
  guard (all (≤x) ys' ∧ all (≥x) zs')) >>
  slowsort ys >>= λ ys' →            ⊇ partition x xs
  slowsort zs >>= λ zs'→
  return (ys' ++ [x] ++ zs')
```

Quicksort for Lists

slowsort xs ⊇ return (quicksort xs)

```
quicksort [ ]      = [ ]
quicksort (x:xs) =
   let (ys, zs) = partition x xs
   in quicksort ys ++ [ x ] ++ quicksort zs
```

Commuting Guards

Definition: m and n commute if

$$m \gg\!= \lambda\, x \to n \gg\!= \lambda\, y \to f\, x\, y \quad = \quad n \gg\!= \lambda\, y \to m \gg\!= \lambda\, x \to f\, x\, y$$

Theorem: guard commutes with other terms.

To prove the commutativity we need:

left zero $\varnothing \gg= f = \varnothing$

left distr. $(m_1 \square m_2) \gg= f = (m_1 \gg= f) \square (m_2 \gg= f)$

right zero $m \gg \varnothing = \varnothing$

right distr. $m \gg= (\lambda\, x \to f_1\, x \square f_2\, x) = (m \gg= f_1) \square (m \gg= f_2)$

Arrays

types

Idx -- index to a global array
e -- type of elements in the array

operators

read :: Idx → m e
write :: Idx → e → m ( )

Arrays

types

Idx -- index to a global array
e -- type of elements in the array

operators

```
read  :: Idx → m e
write :: Idx → e → m ()
```

induced operators

```
readList  :: Idx → Nat → m [e]
writeList :: Idx → [e] → m ()
swap      :: Idx → Idx → m ()
```

Imperative Quicksort

type $\text{iqsort} :: \text{Idx} \to \text{Nat} \to m\,()$

specification $\text{writeList}\ i\ xs \gg \text{iqsort}\ i\ (\#xs) \subseteq$
$\text{slowsort}\ xs \gg\!= \text{writeList}\ i$

#xs: length of xs

Imperative Quicksort

type $\text{iqsort} :: \text{Idx} \to \text{Nat} \to m\,()$

specification $\underline{\text{writeList}\ i\ xs} \gg \text{iqsort}\ i\ (\#xs) \subseteq$

$\text{slowsort}\ xs \gg\!= \text{writeList}\ i$

Imperative Quicksort

type $\text{iqsort} :: \text{Idx} \to \text{Nat} \to m\,(\,)$

specification $\underline{\text{writeList } i \text{ } xs} \gg \underline{\text{iqsort } i \text{ } (\#xs)} \subseteq$
$\text{slowsort } xs \gg\!= \text{writeList } i$

Imperative Quicksort

type $\quad$ iqsort :: Idx → Nat → m ( )

specification $\quad$ writeList i xs ≫ iqsort i (#xs) ⊆
slowsort xs >>= writeList i

Imperative Quicksort

type iqsort :: Idx → Nat → m ( )

specification

precondition

$\boxed{\text{writeList } i \; xs} \gg \text{iqsort } i \; (\#xs) \subseteq$

$\text{slowsort } xs \gg\!= \text{writeList } i$

Imperative Quicksort

type $\text{iqsort} :: \text{Idx} \to \text{Nat} \to m\ (\,)$

specification

precondition

$$\boxed{\text{writeList } i \ xs} \gg \text{iqsort } i\ (\#xs) \subseteq$$
$$\boxed{\text{slowsort } xs \gg\!= \text{writeList } i}$$

postcondition

Imperative Quicksort

type $\quad$ iqsort :: Idx → Nat → m ( )

specification

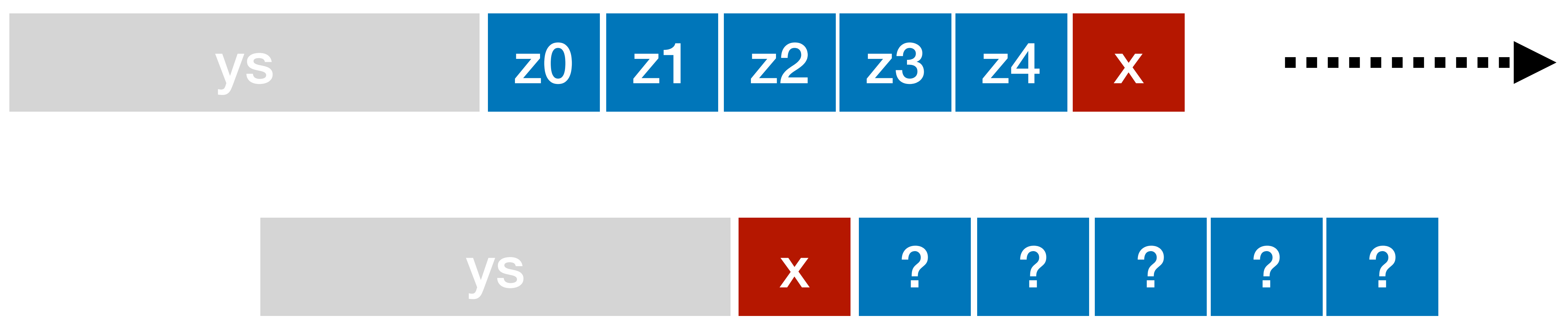
precondition $\qquad$ code to derive

$$\boxed{\text{writeList } i\ xs} \gg \boxed{\text{iqsort } i\ (\#xs)} \subseteq$$
$$\boxed{\text{slowsort } xs \gg\!= \text{writeList } i}$$

postcondition

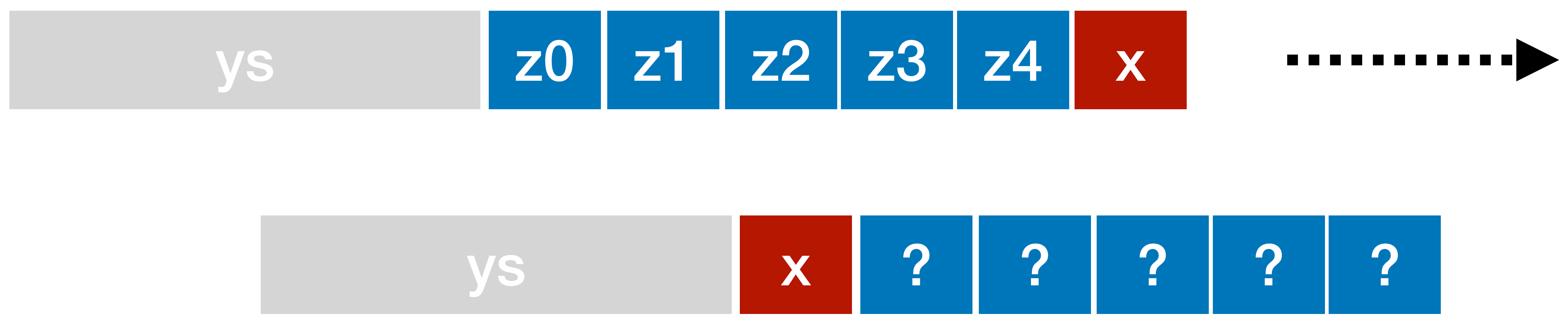
Constructing Imperative Programs

$$\begin{array}{l} \mathit{writeList}\ i\ (ys \mathbin{+\!\!+} zs \mathbin{+\!\!+} [x]) \gg \\ \quad {\color{red}??} \subseteq \\ \mathit{perm}\ zs \gg\!= \lambda\, zs' \rightarrow \\ \quad \mathit{writeList}\ i\ (ys \mathbin{+\!\!+} [x] \mathbin{+\!\!+} zs') \end{array}$$



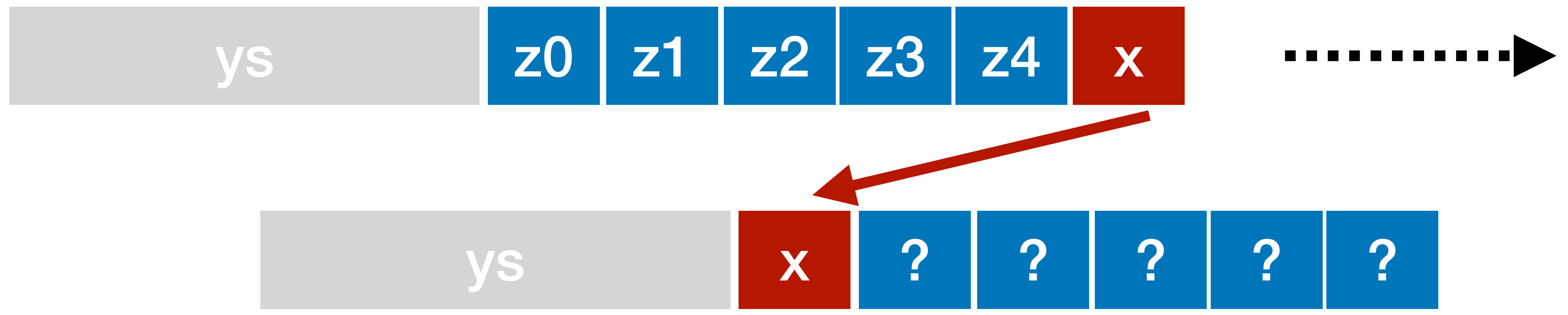
Constructing Imperative Programs

$$\mathit{writeList}\ i\ (\underline{ys \mathbin{+\!\!+} zs \mathbin{+\!\!+} [x])} \gg$$
$$\mathbf{??} \subseteq$$
$$\mathit{perm}\ zs \gg\!\!= \lambda\, zs' \rightarrow$$
$$\mathit{writeList}\ i\ (ys \mathbin{+\!\!+} [x] \mathbin{+\!\!+} zs')$$



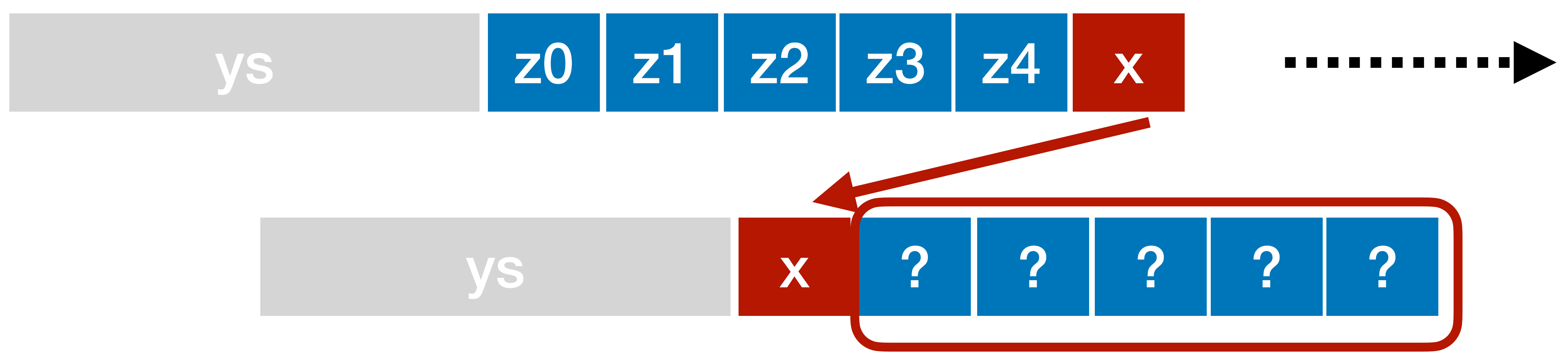
Constructing Imperative Programs

$$\begin{array}{l} \text{writeList } i\ (\underline{ys \mathbin{+\!\!+} zs \mathbin{+\!\!+} [x]}) \gg \\ \quad {\color{red}??} \subseteq \\ \text{perm } zs \gg\!= \lambda\, zs' \to \\ \quad \text{writeList } i\ (\underline{ys \mathbin{+\!\!+} [x] \mathbin{+\!\!+} zs'}) \end{array}$$



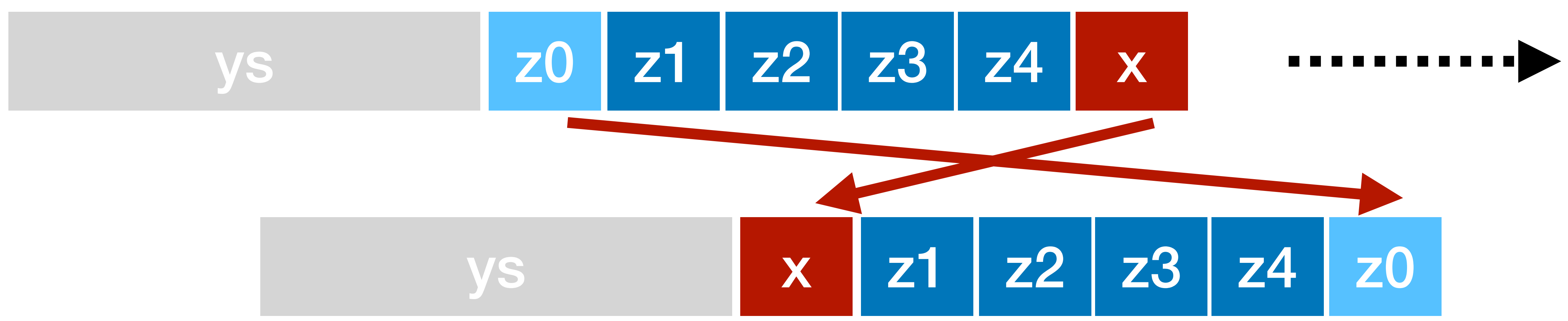
Constructing Imperative Programs

$$
\begin{aligned}
&\mathit{writeList}\ i\ (\underline{ys \mathbin{+\!\!+} zs \mathbin{+\!\!+} [x]}) \gg \\
&\quad \color{red}{??} \subseteq \\
&\mathit{perm}\ zs \gg\!\!= \lambda\, zs' \rightarrow \\
&\quad \mathit{writeList}\ i\ (\underline{ys \mathbin{+\!\!+} [x] \mathbin{+\!\!+} zs'})
\end{aligned}
$$



Constructing Imperative Programs

$$
\begin{array}{l}
\mathit{writeList}\ i\ (ys \mathbin{+\!\!+} zs \mathbin{+\!\!+} [x]) \gg \\
\quad \mathit{swap}\ (i + \#ys)\ (i + \#ys + \#zs) \quad \subseteq \\
\quad \mathit{perm}\ zs \gg\!= \lambda\, zs' \rightarrow \\
\qquad \mathit{writeList}\ i\ (ys \mathbin{+\!\!+} [x] \mathbin{+\!\!+} zs')
\end{array}
$$



Quicksort for Arrays

```
iqsort i 0 = {( )}
iqsort i n =
    read i ≫= λ p →
    ipartition p (i+1) (0,0,n-1) ≫= λ (ny,nz) →
    swap i (i + ny)
    iqsort i ny ≫ iqsort (i+ny+1) nz
```

Conclusions

Monad: a good choice as a calculus for program derivation that involves non-determinism.

Able to apply familiar techniques --- pattern matching, induction on structures or on sizes, etc.

Other effects can be naturally integrated.

Thank you.