liquid resource types for verification and synthesis

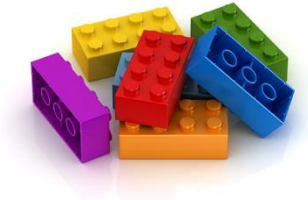
Nadia Polikarpova

with Tristan Knoth, Di Wang, and Jan Hoffmann

UCSDCSE
Computer Science and Engineering

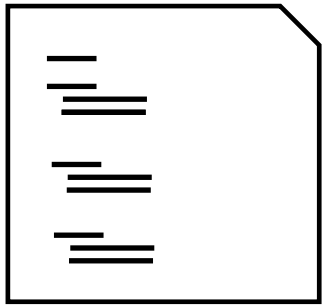
program synthesis

specification

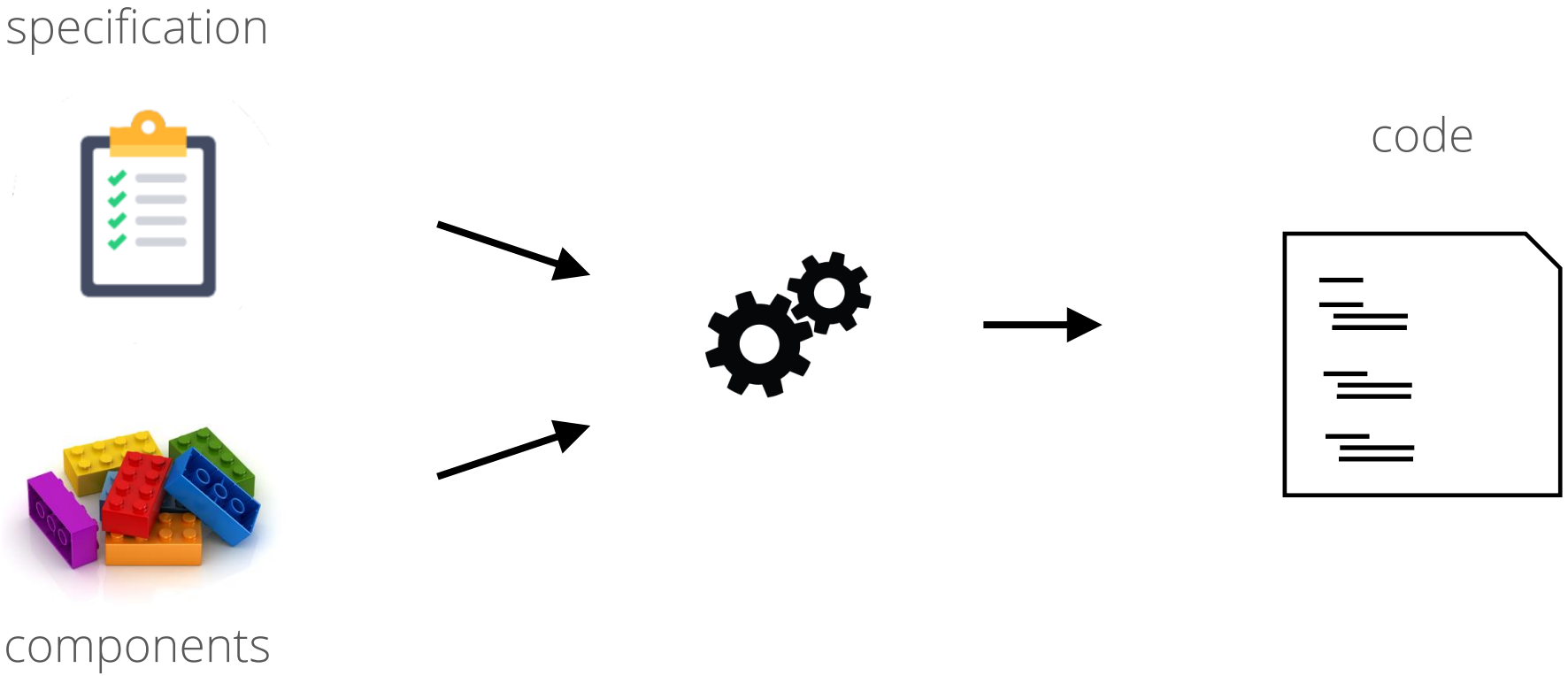


components

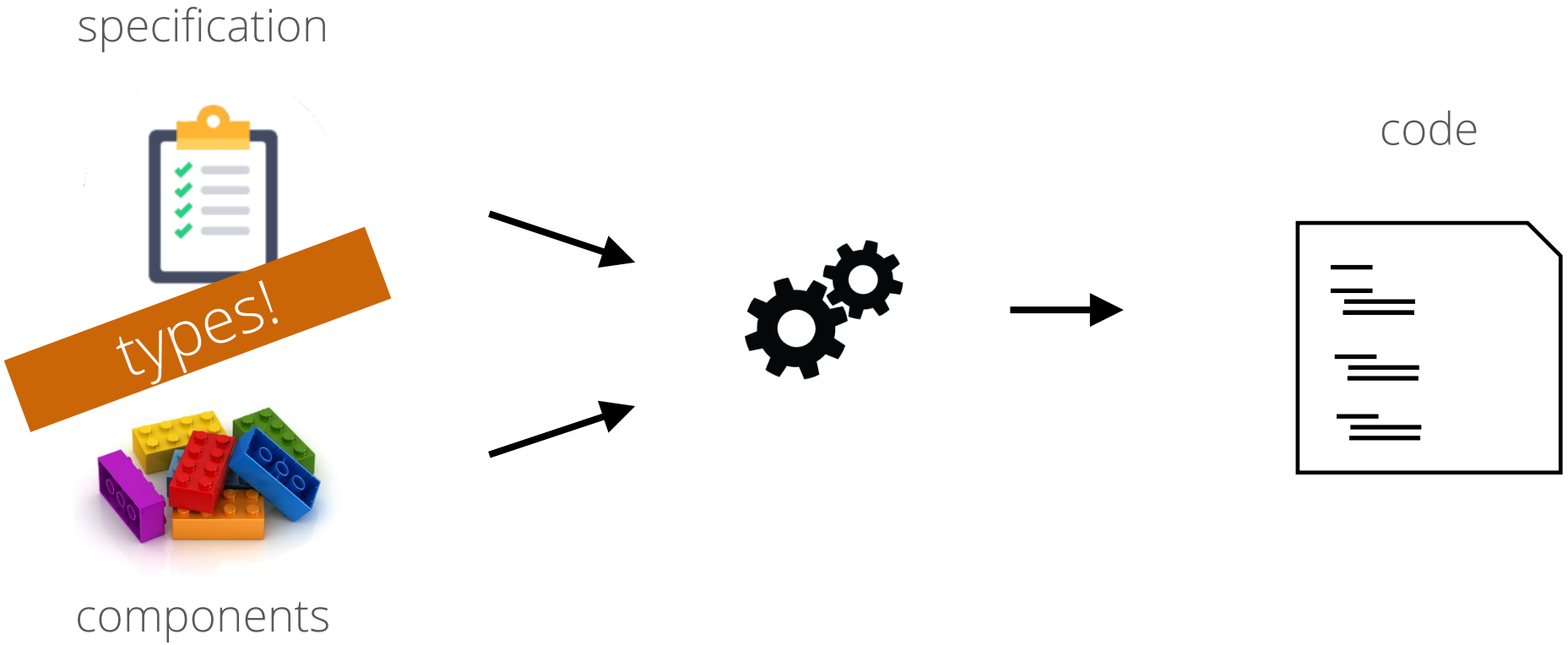
code



type-driven program synthesis



type-driven program synthesis



type-driven program synthesis

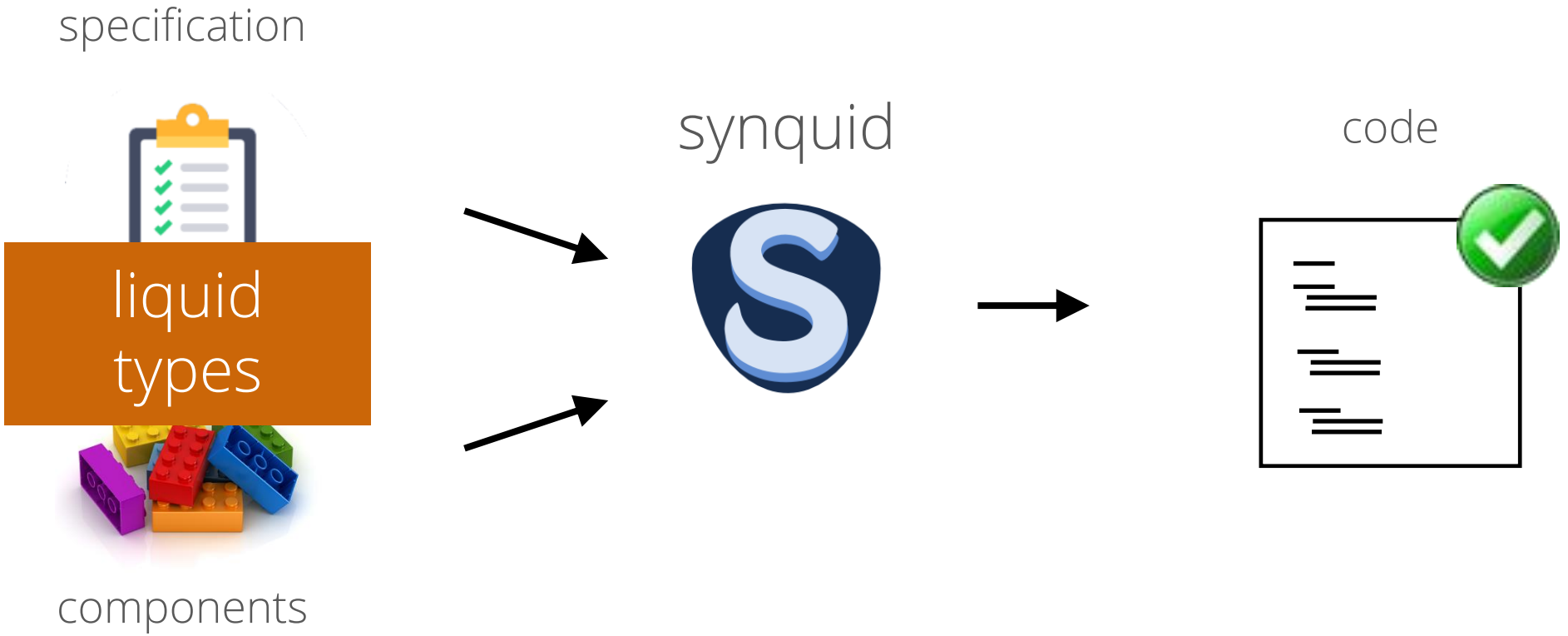
specification

components

synquid

code

type-driven program synthesis



example: compress a list

example: compress a list

input:

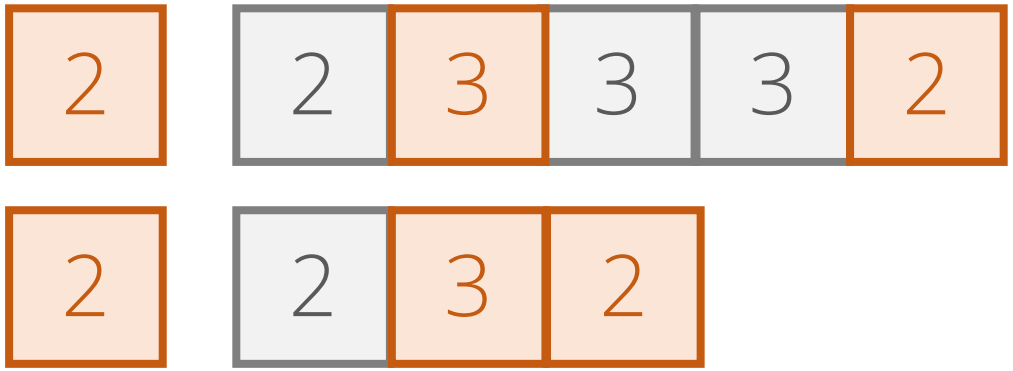
example: compress a list

input:

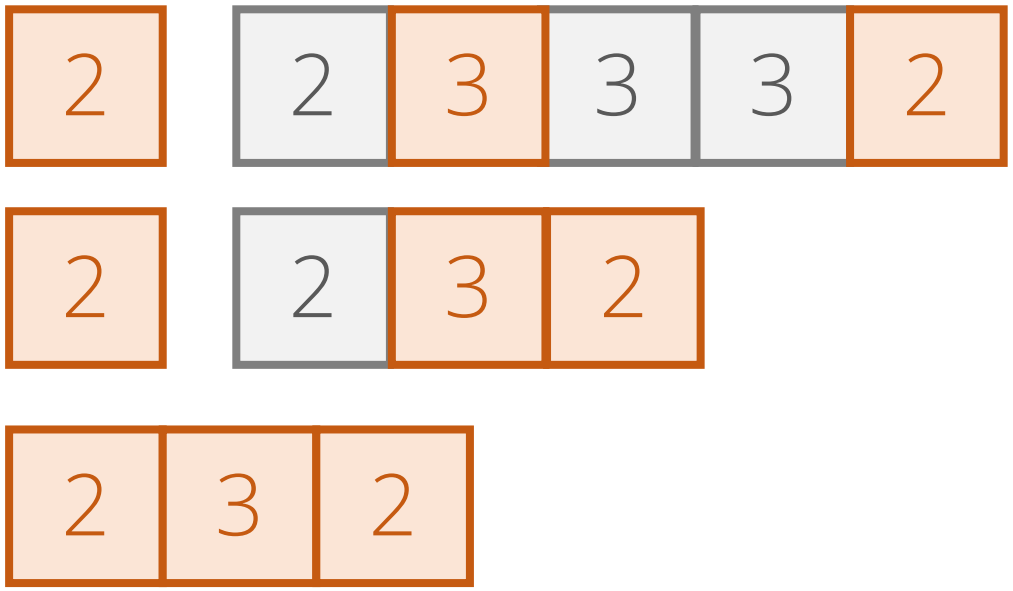
output:

example: compress a list

example: compress a list



example: compress a list

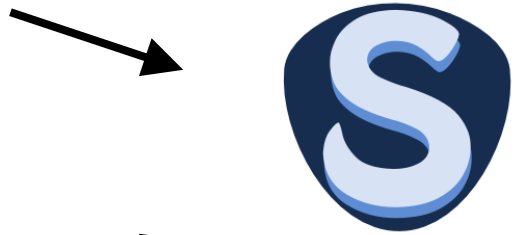


synthesizing compress

specification

?

components



code

?

compress: specification

```
compress :: xs: List a  →  List a
```

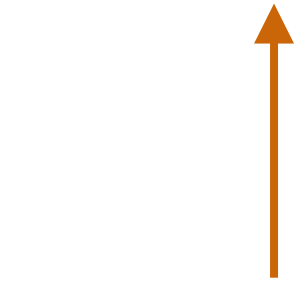
compress: specification

```
compress :: xs: List a  →  CList a
```

compressed list
(no adjacent duplicates)

compress: specification

```
compress :: xs: List a  →  {v:CList a | elems v = elems xs}
```



same set of elements

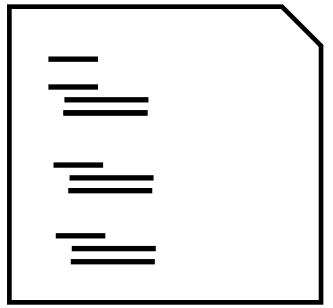
synthesizing compress

specification

```
compress ::
  xs:List a  →
    {v:CList a |
      elems v = elems xs}
```

synquid

code



```
(==), List, CList
```

components

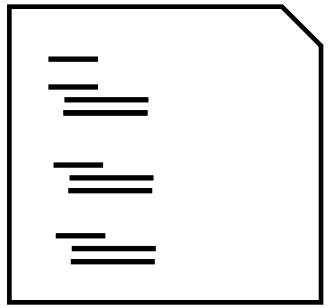
synthesizing compress

specification

```
compress ::
  xs:List a  →
    {v:CList a |
      elems v = elems xs}
```

synquid

code



```
(==), List, CList
```

components

http://comcom.csail.mit.edu/demos/#compress

compress: generated solution

```
compress xs =
  match xs with
    Nil → Nil
    Cons y ys →
      match compress ys with
        Nil → Cons y Nil
        Cons z zs → if z == y
                      then compress ys
                      else Cons y (Cons z zs)
```

compress: generated solution

```
compress xs =
  match xs with
    Nil → Nil
    Cons y ys →
      match compress ys with
        Nil → Cons y Nil
        Cons z zs → if z == y
                      then compress ys
                      else Cons y (Cons z zs)
```

exponential!

compress: generated solution

```
compress xs =
  match xs with
    Nil → Nil
    Cons y ys →
      match compress ys with
        Nil → Cons y Nil
        Cons z zs → if z == y
                      then compress ys
                      else Cons y (Cons z zs)
```

exponential!

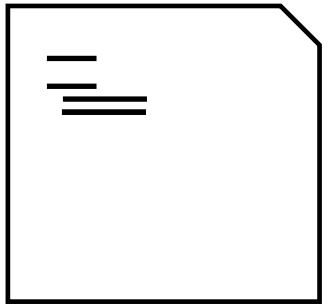
synthesizing efficient programs

specification

code

we have:

compress a list



$O(2^{|xs|})$

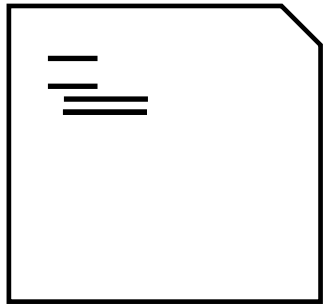
synthesizing efficient programs

specification

code

we have:

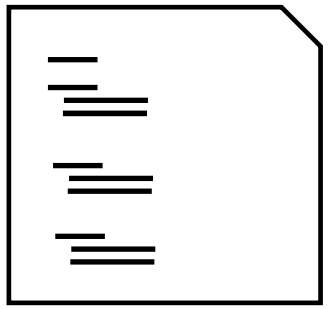
compress a list



$O(2^{|xs|})$

we want:

compress a list
in linear time



$O(|xs|)$

synthesizing efficient programs

specification

code

compress a list
in linear time

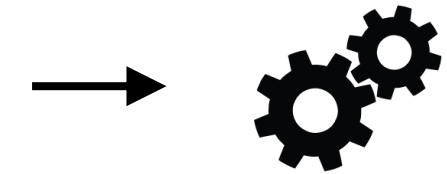
$O(|xs|)$

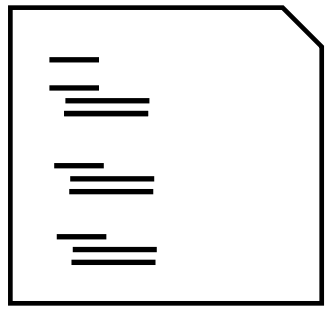
synthesizing efficient programs

specification

code

liquid resource types





O(|xs|)

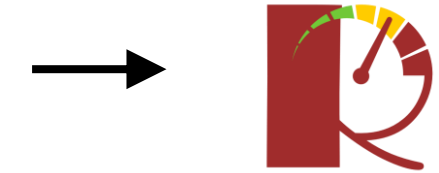
synthesizing efficient programs

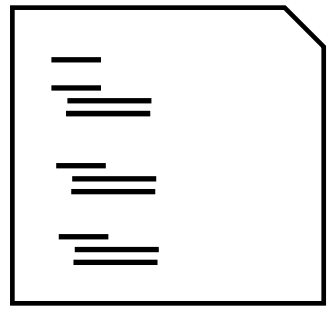
specification

resyn

code

liquid resource types





O(|xs|)

this talk

1. liquid types + resource bounds

this talk

1. liquid types + resource bounds
2. type checking

this talk

1. liquid types + resource bounds
2. type checking
3. non-linear bounds

this talk

1. liquid types + resource bounds
2. type checking
3. non-linear bounds
4. value-dependent bounds

this talk

1. liquid types + resource bounds
2. type checking
3. non-linear bounds
4. value-dependent bounds

[PLDI'19]

this talk

1. liquid types + resource bounds
2. type checking

[PLDI'19]

3. non-linear bounds

[ICFP'20 ?]

4. value-dependent bounds

this talk

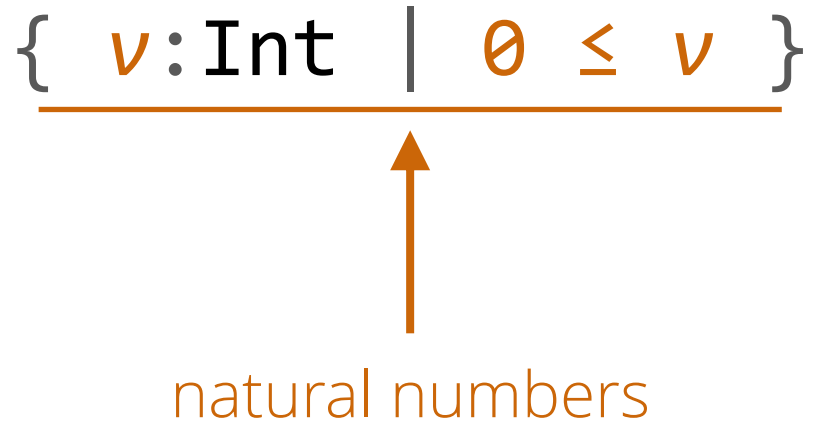
1. liquid types + resource bounds
2. type checking
3. non-linear bounds
4. value-dependent bounds

types + refinements

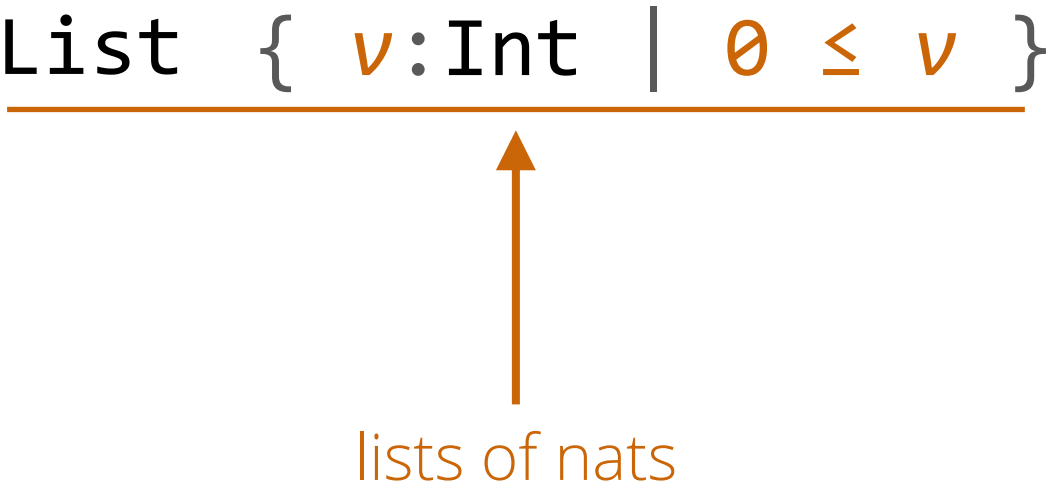
`Int`

types + refinements

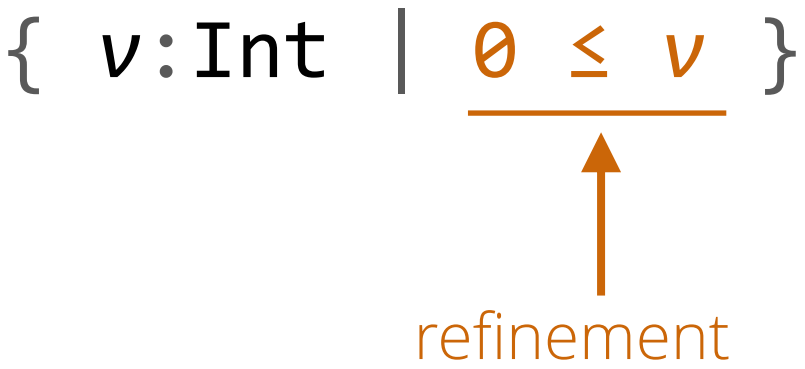
types + refinements



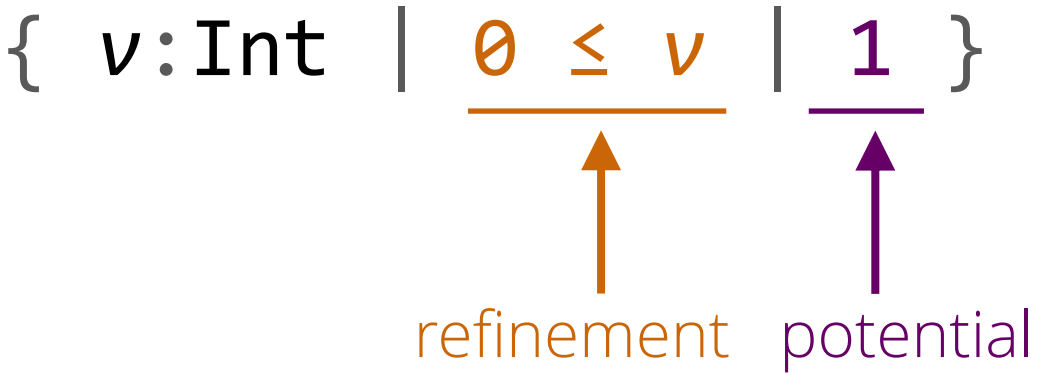
types + refinements



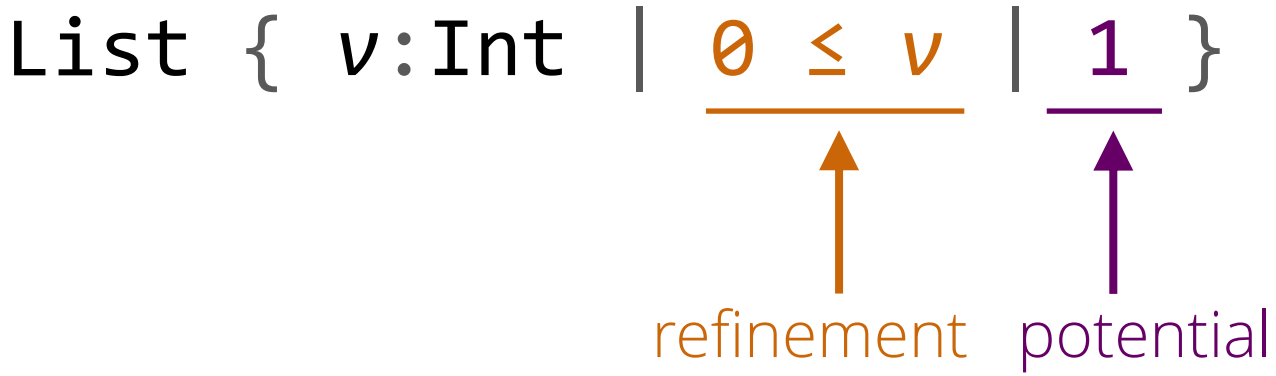
types + refinements + resources



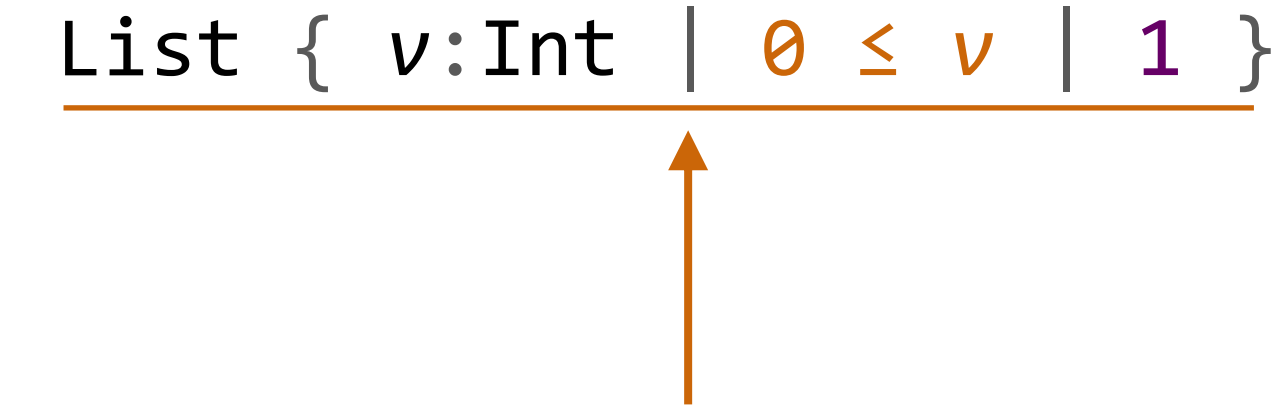
types + refinements + resources



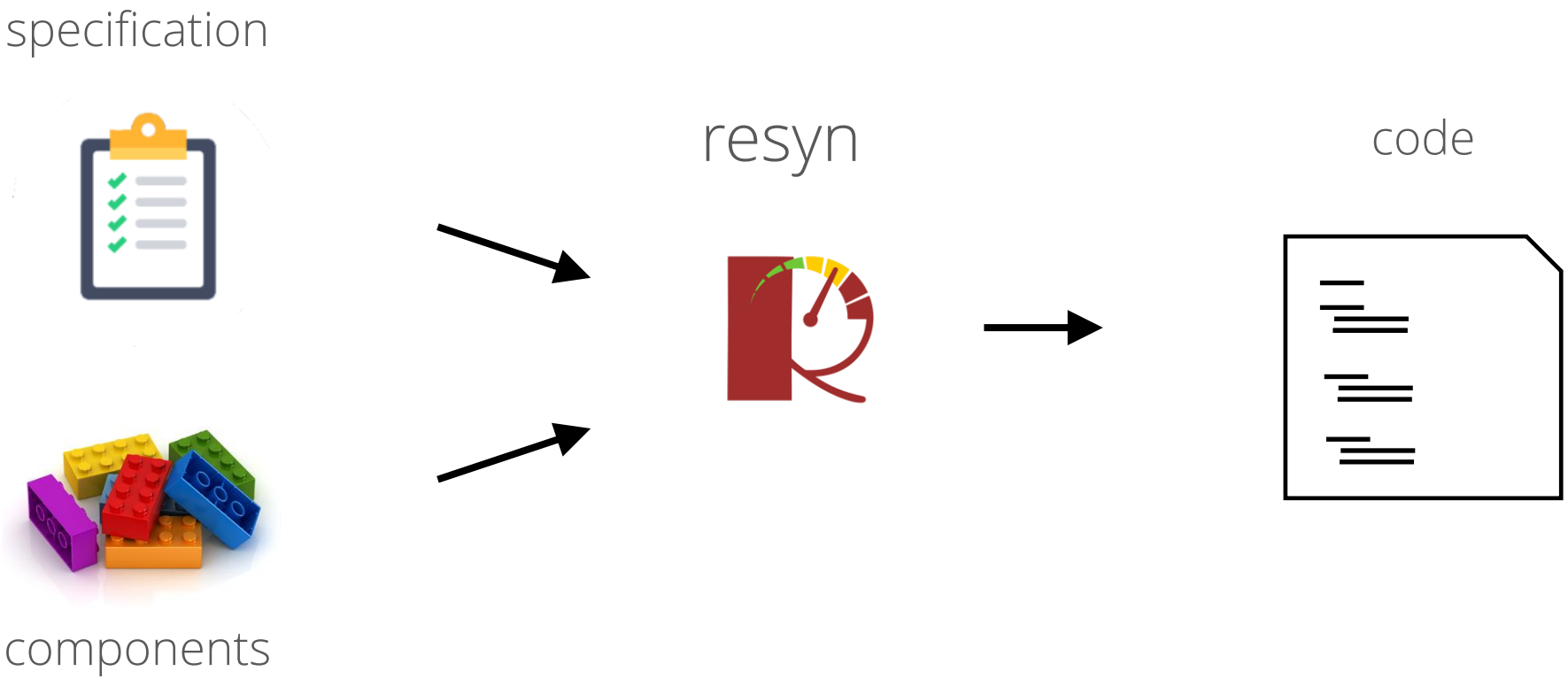
types + refinements + resources



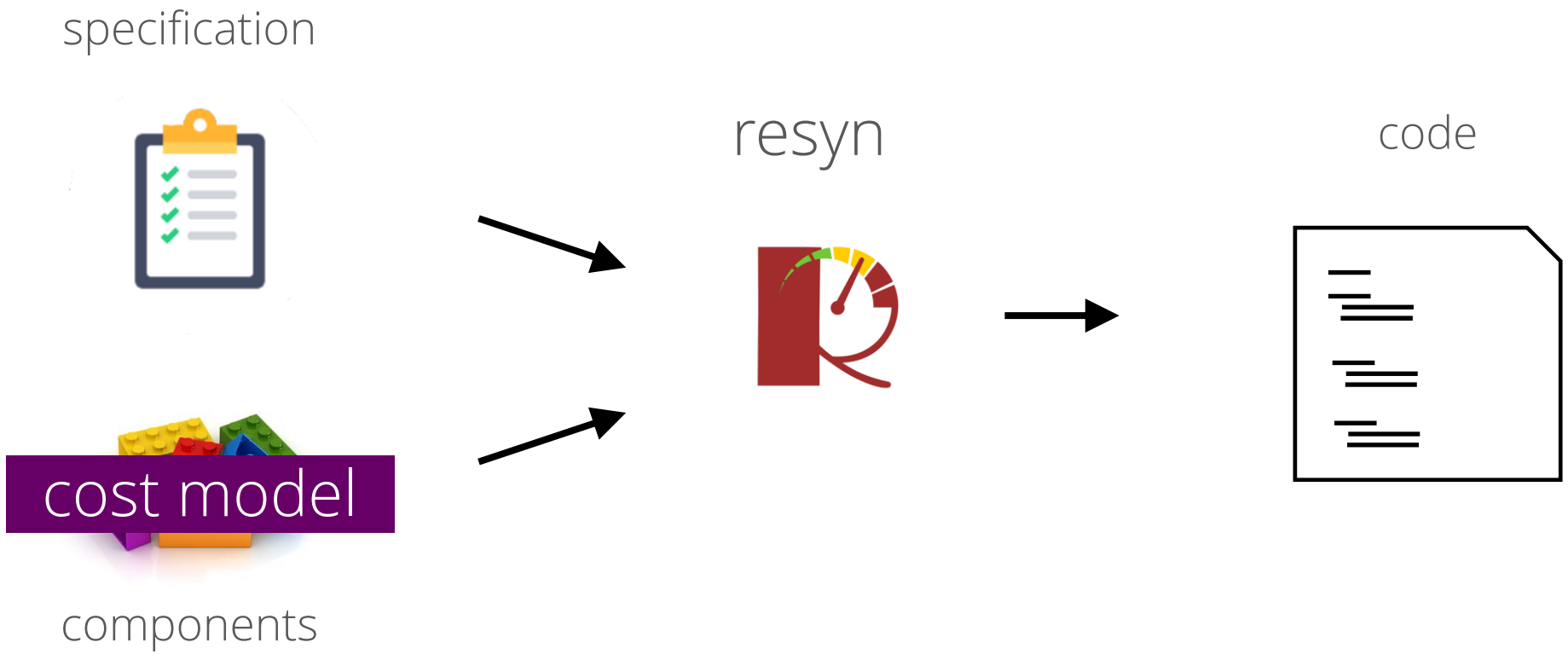
types + refinements + resources



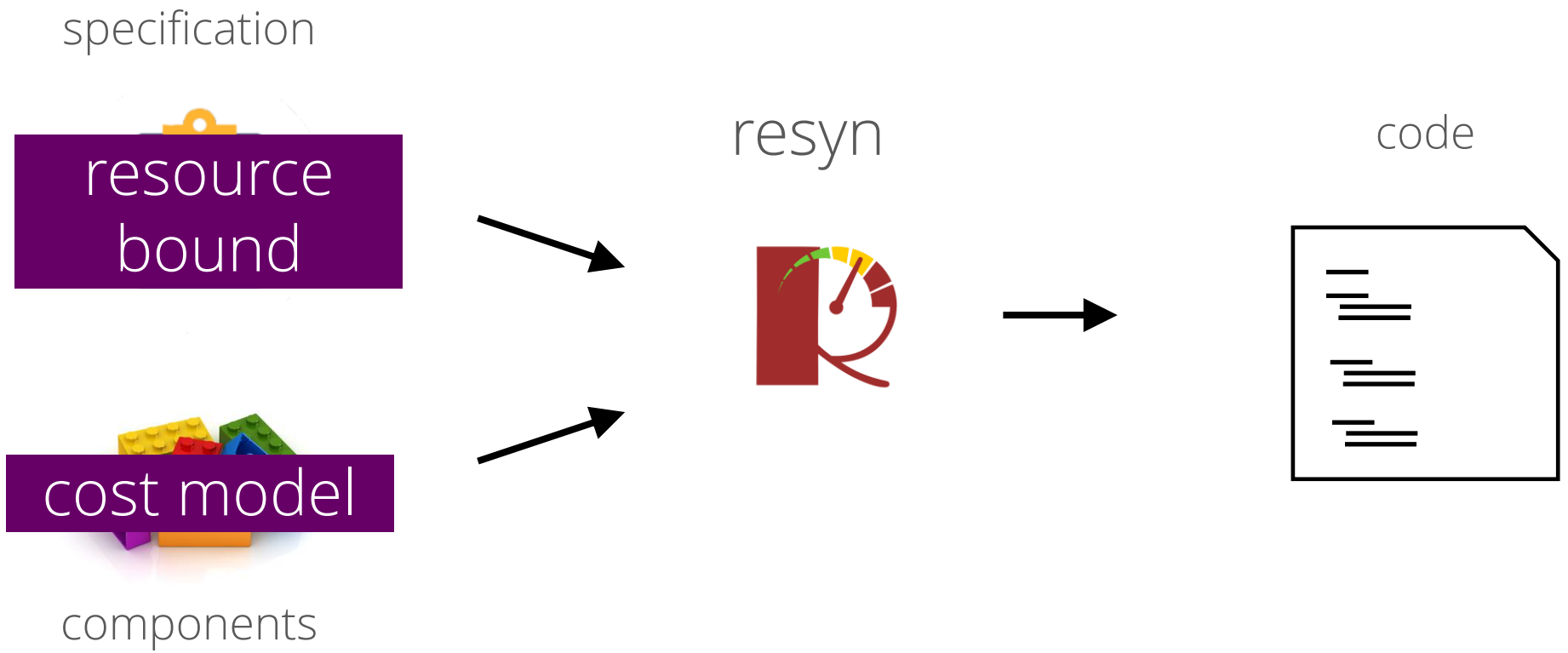
synthesizing efficient programs



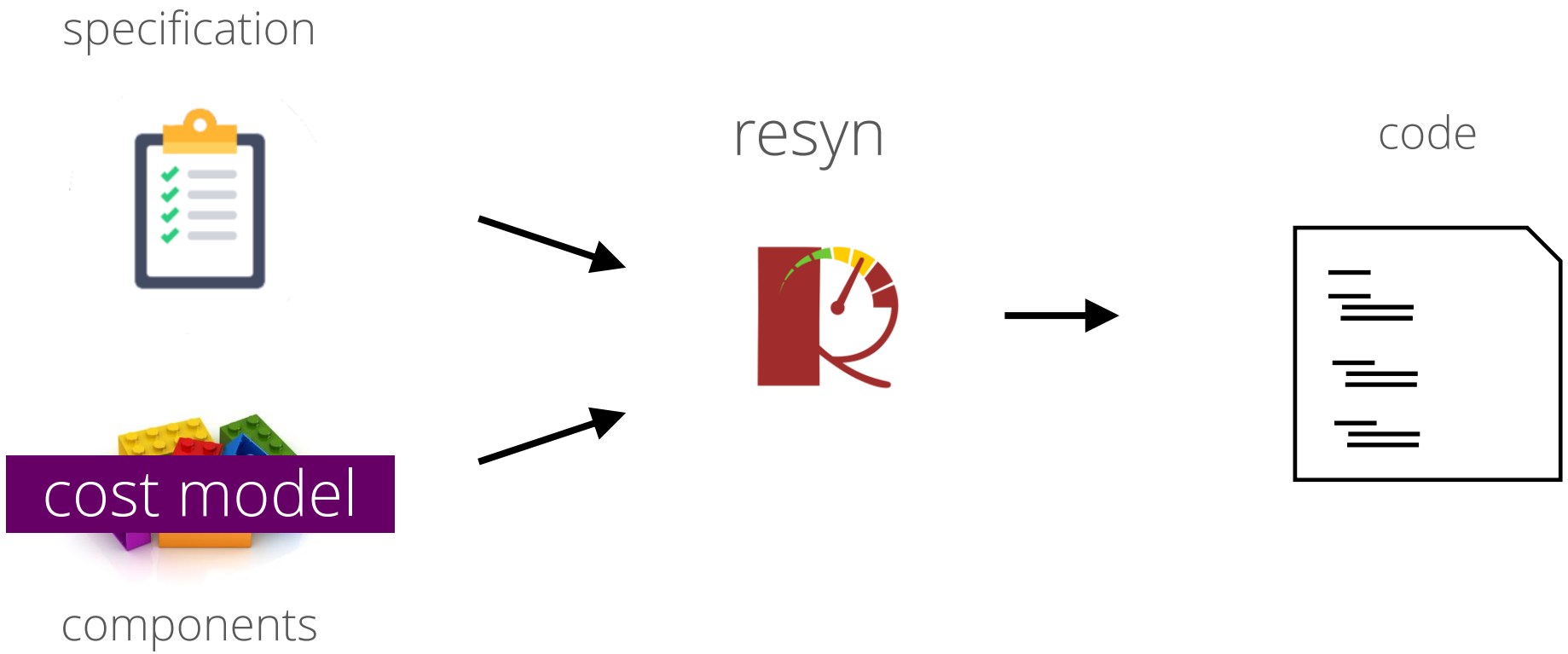
synthesizing efficient programs



synthesizing efficient programs



synthesizing efficient programs



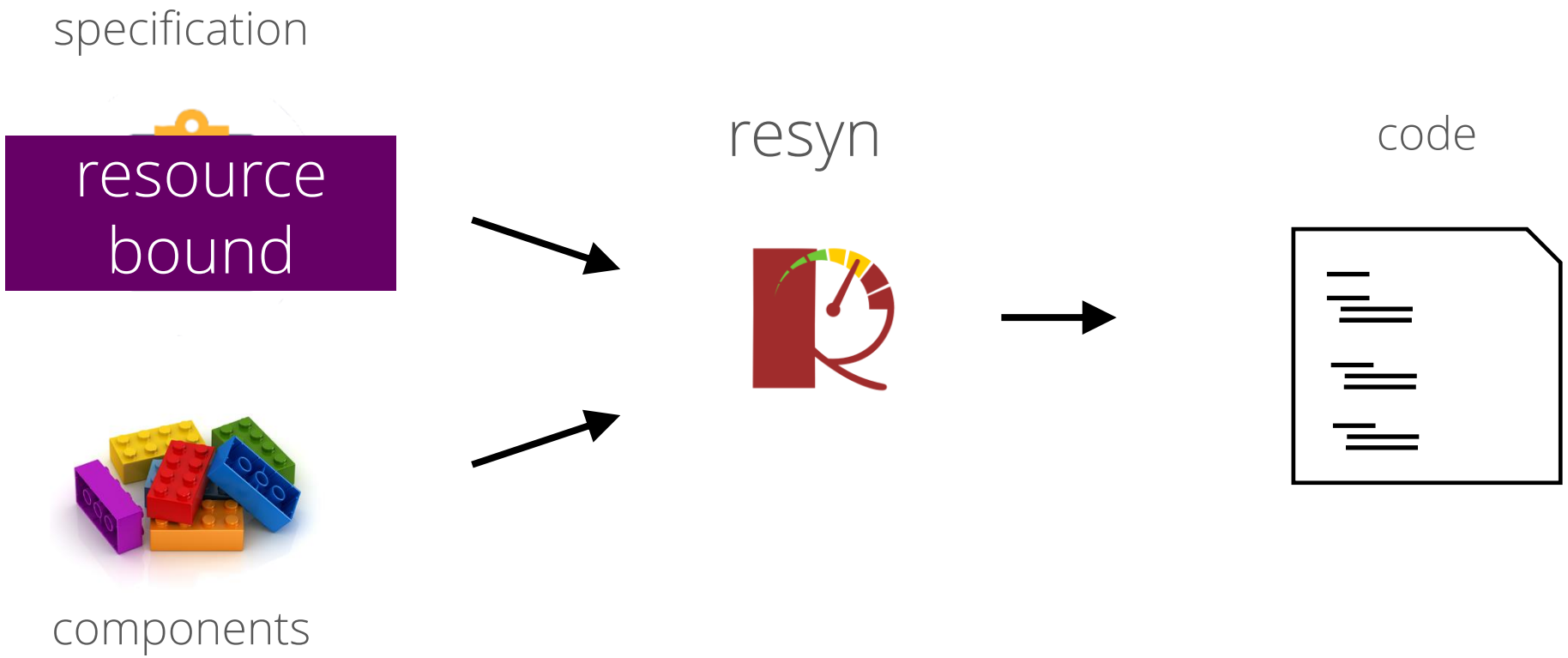
compress: cost model

```
(==) :: x:a  →  y:a →
                {v:Bool | v = (x == y)}
```

compress: cost model

```
(==) :: x:a  →  y:{a| |1}  →
                {ν:Bool | ν = (x == y)}
```

synthesizing efficient programs



compress: resource bound

```
compress :: xs:List a                    →
            {v:CList a | elems v = elems xs}
```

compress: resource bound

```
compress :: xs:List {a | | 1}  →
                {v:CList a | elems v = elems xs}
```

synthesizing linear compress

specification

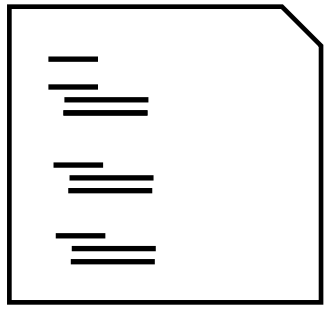
```
compress ::
  xs:List {a | | 1}  →
    {v:CList a | …}
```

```
(==) :: x:a  →  y:{a | | 1}
       →  {v:Bool | …}
```

components

resyn

code



O(|xs|)

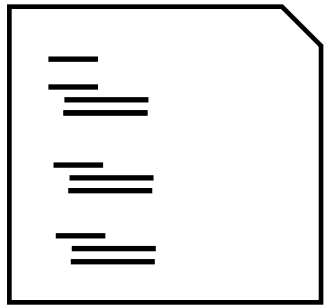
synthesizing linear compress

specification

```
compress ::
  xs:List {a | | 1}  →
    {v:CList a | …}
```

resyn

code



O(|xs|)

```
(==) :: x:a  →  y:{a | | 1}
        →  {v:Bool | …}
```

components

http://comcom.csail.mit.edu/demos/#compress_linear

this talk

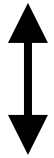
1. liquid types + resource bounds
2. type checking
3. non-linear bounds
4. value-dependent bounds

checking compress (exponential)

```
compress :: List {a||1} → List a
```

checking compress (*exponential*)

```
compress :: List {a||1} → List a
```



```
compress xs =
  match xs with
    Nil → Nil
    Cons y ys →
      match compress ys with
        Nil → Cons y Nil
        Cons z zs → if z == y
                    then compress ys
                    else …
```

checking compress (*exponential*)

```
List a
```

```
compress xs =
  match xs with
    Nil → Nil
    Cons y ys →
      match compress ys with
        Nil → Cons y Nil
        Cons z zs → if z == y
                    then compress ys
                    else …
```

Context:

```
compress: List {a||1} → List a
xs: List {a||1}
```

checking compress (*exponential*)

```
List a
```

Context:

```
compress: List {a||1} → List a
xs: List {a||1}
```

idea: generate constraints that have a solution if

```
compress xs =
  match xs with
    Nil → Nil
    Cons y ys →
      match compress ys with
        Nil → Cons y Nil
        Cons z zs → if z == y
                      then compress ys
                      else …
```

checking compress (*exponential*)

```
List a
```

```
compress xs =
  match xs with
    Nil → Nil
    Cons y ys →
      match compress ys with
        Nil → Cons y Nil
        Cons z zs → if z == y
                    then compress ys
                    else …
```

Context:

```
compress: List {a||1} → List a
xs: List {a||1}
```

idea: generate constraints that have a solution if we can partition resources in context

checking compress (exponential)

```
List a
```

```
compress xs =
  match xs with
    Nil → Nil
    Cons y ys →
      match compress ys with
        Nil → Cons y Nil
        Cons z zs → if z == y
                    then compress ys
                    else …
```

Context:

```
compress: List {a||1} → List a
xs: List {a||1}
```

idea: generate constraints that have a solution if we can partition resources in context between terms that require potential

checking compress (*exponential*)

```
List a
```

```
compress xs =
  match xs with
    Nil → Nil
    Cons y ys →
      match compress ys with
        Nil → Cons y Nil
        Cons z zs → if z == y
                    then compress ys
                    else …
```

Context:

```
compress: List {a||1} → List a
xs: List {a||1}
```

checking compress (*exponential*)

Context:

```
compress: List {a||1} → List a
xs: List {a||1}
```

```
List a
```

```
compress xs =
  match xs with
    Nil → Nil
    Cons y ys →
      match compress ys with
        Nil → Cons y Nil
        Cons z zs → if z == y
                      then compress ys
                      else …
```

checking compress (*exponential*)

```
List a
```

```
compress xs =
  match xs with
    Nil → Nil
    Cons y ys →
      match compress ys with
        Nil → Cons y Nil
        Cons z zs → if z == y
                    then compress ys
                    else …
```

Context:

```
compress: List {a||1} → List a
xs: List {a||1}
```

checking compress (exponential)

```
List a
```

```
compress xs =
  match xs with
    Nil → Nil
    Cons y ys →
      match compress ys with
        Nil → Cons y Nil
        Cons z zs → if z == y
                      then compress ys
                      else …
```

Context:

```
compress: List {a||1} → List a
xs: List {a||1}
```

```
compress: …
xs: List {a||???}
```

+

```
compress: …
xs: List {a||???}
```

checking compress (exponential)

```
List a
```

```
compress xs =
  match xs with
    Nil → Nil
    Cons y ys →
      match compress ys with
        Nil → Cons y Nil
        Cons z zs → if z == y
                    then compress ys
                    else …
```

Context:

```
compress: List {a||1} → List a
xs: List {a||1}
```

```
compress: …
xs: List {a||1}
```

+

```
compress: …
xs: List {a||0}
```

checking compress (*exponential*)

List a

```
compress xs =
  match xs :: List {a||1} with
    Nil → Nil
    Cons y ys →
      match compress ys with
        Nil → Cons y Nil
        Cons z zs → if z == y
                    then compress ys
                    else …
```

```
compress: …
xs: List {a||1}
```

checking compress (*exponential*)

```
List a
```

```
compress xs =
  match xs :: List {a||1} with
    Nil → Nil
    Cons y ys →
      match compress ys with
        Nil → Cons y Nil
        Cons z zs → if z == y
                    then compress ys
                    else …
```

```
compress: …
xs: List {a||1}
```

destructing yields

```
y: {a||1}   ys: List {a||1}
```

checking compress (exponential)

```
List a
```

```
compress xs =
  match xs with
    Nil → Nil
    Cons y ys →
      match compress ys with
        Nil → Cons y Nil
        Cons z zs → if z == y
                    then compress ys
                    else …
```

Context:

```
compress: List {a||1} → List a
ys: List {a||1}
y: {a||1}
```

checking compress (exponential)

```
List a
```

```
compress xs =
  match xs with
    Nil → Nil
    Cons y ys →
      match compress ys with
        Nil → Cons y Nil
        Cons z zs → if z == y
                      then compress ys
                      else …
```

Context:

```
compress: List {a||1} → List a
ys: List {a||1}
y: {a||1}
```

checking compress (exponential)

List a

```
compress xs =
  match xs with
    Nil → Nil
    Cons y ys →
      match compress ys with
        Nil → Cons y Nil
        Cons z zs → if z == y
                    then compress ys
                    else …
```

Context:
compress: List {a||1} → List a
ys: List {a||1}
y: {a||1}

ys: List {a||*p*}

+

ys: List {a||*q*}

checking compress (exponential)

```
List a
```

Context:

```
compress: List {a||1} → List a
ys: List {a||1}
y: {a||1}
```

```
compress xs =
  match xs with
    Nil → Nil
    Cons y ys →
      match compress (ys :: List {a||p}) with
        Nil → Cons y Nil
        Cons z zs → if z == y
                      then compress (ys :: List {a||q})
                      else …
```

checking compress (exponential)

```
List a
```

```
compress xs =
  match xs with
    Nil → Nil
    Cons y ys →
      match compress (ys :: List {a||p}) with
        Nil → Cons y Nil
        Cons z zs → if z == y
                      then compress (ys :: List {a||q})
                      else …
```

Context:

```
compress: List {a||1} → List a
ys: List {a||1}
y: {a||1}
```

Constraints: ∃*p*,*q*:

$$1 = p + q$$

1. total potential must be partitioned

checking compress (exponential)

```
List a
```

```
compress xs =
  match xs with
    Nil → Nil
    Cons y ys →
      match compress (ys :: List {a||p}) with
        Nil → Cons y Nil
        Cons z zs → if z == y
                      then compress (ys :: List {a||q})
                      else …
```

Context:

```
compress: List {a||1} → List a
ys: List {a||1}
y: {a||1}
```

Constraints: ∃p,q:

$$1 = p + q$$

$$p \geq 1$$

2. p must be enough to call compress

checking compress (exponential)

```
List a
```

```
compress xs =
  match xs with
    Nil → Nil
    Cons y ys →
      match compress ys with
        Nil → Cons y Nil
        Cons z zs → if z == y
                      then compress (ys :: List {a||q})
                      else …
```

3. q must be enough to call compress

Context:

```
compress: List {a||1} → List a
ys: List {a||1}
y: {a||1}
```

Constraints: $\exists p, q$:

$$1 = p + q$$

$$p \geq 1$$

$$q \geq 1$$

checking compress (exponential)

```
List a
```

```
compress xs =
  match xs with
    Nil → Nil
    Cons y ys →
      match compress ys with
        Nil → Cons y Nil
        Cons z zs → if z == y
                    then compress ys
                    else …
```

Context:

```
compress: List {a||1} → List a
ys: List {a||1}
y: {a||1}
```

Constraints: ∃p,q:

$$1 = p + q$$

$$p \geq 1$$

$$q \geq 1$$

SMT solver: UNSAT!

checking compress (exponential)

```
List a
```

```
compress xs =
  match xs with
    Nil → Nil
    Cons y ys →
      match compress ys with
        Nil → Cons y Nil
        Cons z zs → if z == y
                    then compress ys
                    else …
```

Context:

```
compress: List {a||1} → List a
ys: List {a||1}
y: {a||1}
```

Constraints: ∃p,q:

$$1 = p + q$$

$$p \geq 1$$

$$q \geq 1$$

SMT solver: UNSAT!

checking compress (linear)

```
List a
```

Context:

```
compress: List {a||1} → List a
ys: List {a||1}
y: {a||1}
```

```
compress xs =
  match xs with
    Nil → Nil
    Cons y ys →
      match compress (ys :: List {a||1}) with
        Nil → Cons y Nil
        Cons z zs → if z == (y :: {a||1})
                      then Cons z zs
                      else Cons y (Cons z zs)
```

checking compress (linear)

```
List a
```

```
compress xs =
  match xs with
    Nil → Nil
    Cons y ys →
      match compress (ys :: List {a||1}) with
        Nil → Cons y Nil
        Cons z zs → if z == (y :: {a||1})
                      then Cons z zs
                      else Cons y (Cons z zs)
```

all potential in ys goes here

Context:

```
compress: List {a||1} → List a
ys: List {a||1}
y: {a||1}
```

checking compress (linear)

```
List a
```

```
compress xs =
  match xs with
    Nil → Nil
    Cons y ys →
      match compress (ys :: List {a||1}) with
        Nil → Cons y Nil
        Cons z zs → if z == (y :: {a||1})
                      then Cons z zs
                      else Cons y (Cons z zs)
```

Context:

```
compress: List {a||1} → List a
ys: List {a||1}
y: {a||1}
```

all potential in ys goes here

pays for comparison

checking compress (linear)

```
List a
```

```
compress xs =
  match xs with
    Nil → Nil
    Cons y ys →
      match compress (ys :: List {a||1}) with
        Nil → Cons y Nil
        Cons z zs → if z == (y :: {a||1})
                    then Cons z zs
                    else Cons y (Cons z zs)
```

Context:

```
compress: List {a||1} → List a
ys: List {a||1}
y: {a||1}
```

all potential in ys goes here

pays for comparison

no potential consumed here

checking compress (linear)

```
List a
```

```
compress xs =
  match xs with
    Nil → Nil
    Cons y ys →
      match compress (ys :: List {a||1}) with
        Nil → Cons y Nil
        Cons z zs → if z == (y :: {a||1})
                    then Cons z zs
                    else Cons y (Cons z zs)
```

Context:

```
compress: List {a||1} → List a
ys: List {a||1}
y: {a||1}
```

all potential in ys goes here

pays for comparison

no potential consumed here

formally: context sharing

$$\frac{}{\Gamma \vdash \textbf{match}\ e\ \textbf{with}\ e_1; \lambda x\ xs.\ e_2\ \ ::\ T}$$

formally: context sharing

$$\Gamma = \Gamma_1 + \Gamma_2$$

$$\frac{}{\Gamma \vdash \textbf{match}\ e\ \textbf{with}\ e_1; \lambda x\ xs.\ e_2\ \ ::\ T}$$

formally: context sharing

$$\Gamma = \Gamma_1 + \Gamma_2$$
$$\Gamma_1 \vdash e :: \text{List } T'$$

$$\frac{}{\Gamma \vdash \textbf{match } e \textbf{ with } e_1; \lambda x\ xs.\ e_2\ :: T}$$

formally: context sharing

$$\Gamma = \Gamma_1 + \Gamma_2$$
$$\Gamma_1 \vdash e :: \text{List } T'$$
$$\Gamma_2 \vdash e_1 :: T$$

$$\frac{}{\Gamma \vdash \textbf{match } e \textbf{ with } e_1; \lambda x\ xs.\ e_2 \ :: T}$$

formally: context sharing

$$\frac{\begin{array}{c}\Gamma = \Gamma_1 + \Gamma_2 \\ \Gamma_1 \vdash e :: \mathrm{List}\, T' \\ \Gamma_2 \vdash e_1 :: T \\ \Gamma_2, x{:}\, T', xs{:}\, \mathrm{List}\, T' \vdash e_1 :: T\end{array}}{\Gamma \vdash \textbf{match}\ e\ \textbf{with}\ e_1; \lambda x\ xs.\, e_2\ \ :: T}$$

formally: subtyping

$$\frac{}{\Gamma \vdash \{B \mid r \mid p\} <: \{B \mid r' \mid p'\}}$$

formally: subtyping

$$\frac{[\![\Gamma]\!] \Rightarrow r \Rightarrow r'}{\Gamma \vdash \{B \mid r \mid p\} <: \{B \mid r' \mid p'\}}$$

formally: subtyping

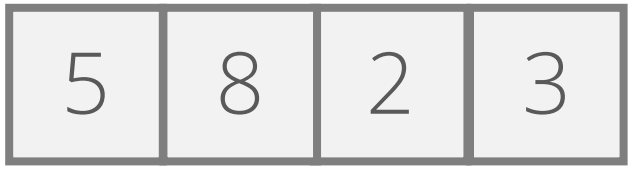
$$\frac{[\![\Gamma]\!] \Rightarrow r \Rightarrow r' \qquad [\![\Gamma]\!] \Rightarrow p \geq p'}{\Gamma \vdash \{B \mid r \mid p\} <: \{B \mid r' \mid p'\}}$$

this talk

1. liquid types + resource bounds
2. type checking
3. non-linear bounds
4. value-dependent bounds

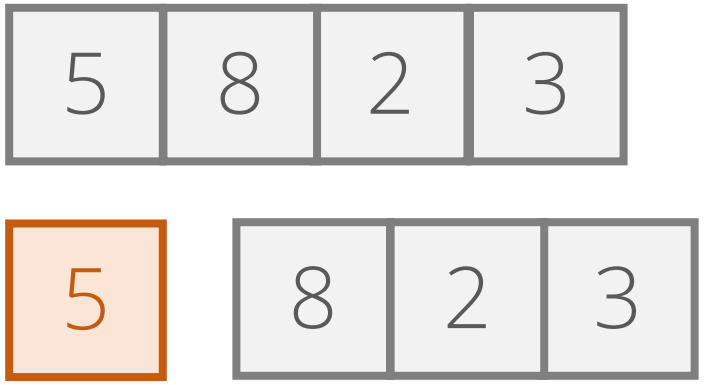
example: insertion sort

input:



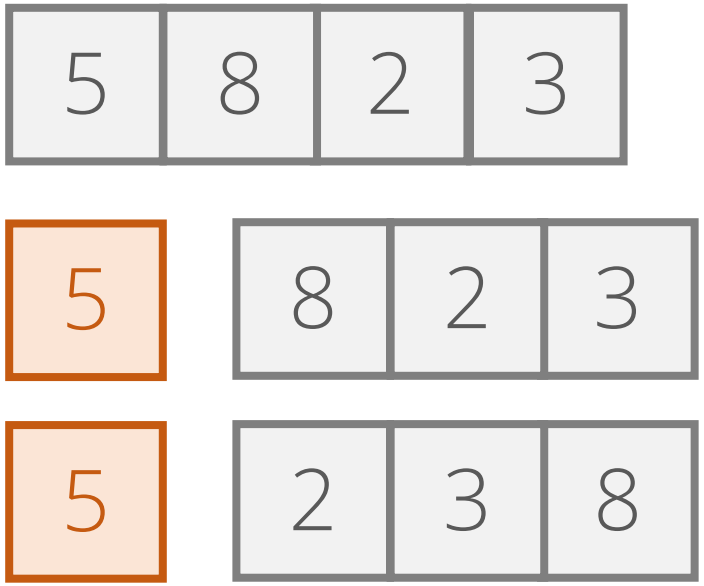
example: insertion sort

input:

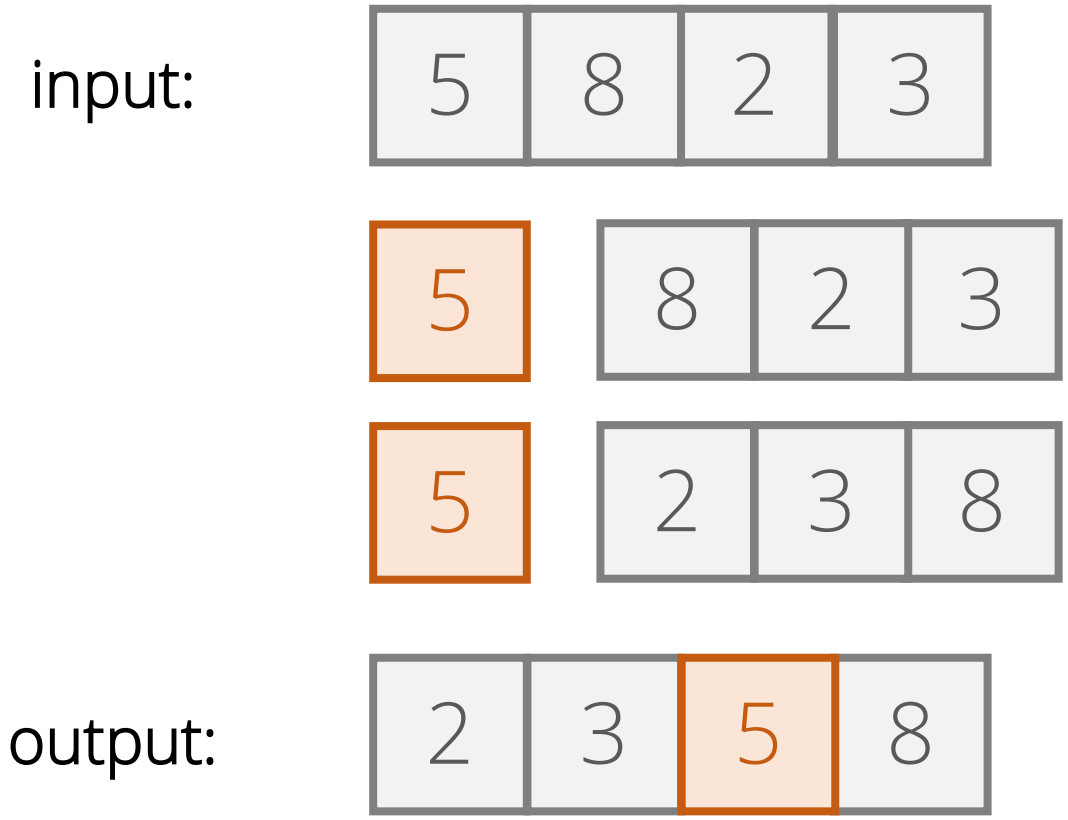


example: insertion sort

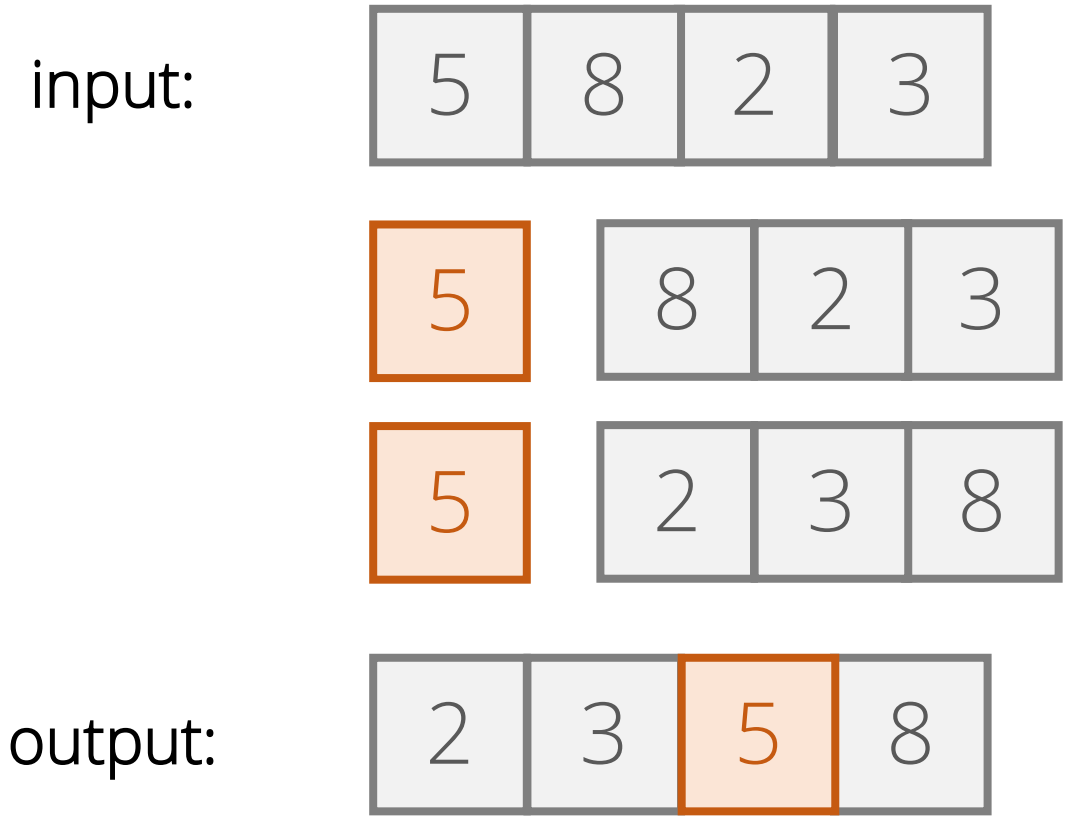
input:



example: insertion sort



example: insertion sort



how many comparisons does insertion sort make?

example: insertion sort

```
sort xs =
  match xs with
    Nil → Nil
    Cons y ys →
      insert y (sort ys)
```

example: insertion sort

```
sort xs =
  match xs with
    Nil → Nil
    Cons y ys →
      insert y (sort ys)
```

```
insert x xs =
  match xs with
    Nil → Cons x Nil
    Cons y ys →
      if x ≤ y
        then Cons x xs
        else Cons y (insert x ys)
```

example: insertion sort

```
(≤) :: x:a  →  y:{a | | 1}  →  Bool
```

```
sort xs =
  match xs with
    Nil → Nil
    Cons y ys →
      insert y (sort ys)
```

```
insert x xs =
  match xs with
    Nil → Cons x Nil
    Cons y ys →
      if x ≤ y
        then Cons x xs
        else Cons y (insert x ys)
```

example: insertion sort

```
sort xs =
  match xs with
    Nil → Nil
    Cons y ys →
      insert y (sort ys)
```

```
(≤) :: x:a  →  y:{a | | 1}  →  Bool

insert :: a → List {a| |1} → List a
insert x xs =
  match xs with
    Nil → Cons x Nil
    Cons y ys →
      if x ≤ y
        then Cons x xs
        else Cons y (insert x ys)
```

example: insertion sort

```
sort xs =
  match xs with
    Nil → Nil
    Cons y ys →
      insert y (sort ys)
```

```
(≤) :: x:a  →  y:{a | | 1}  →  Bool
```

```
insert :: a → List {a| |1} → List a
insert x xs =
  match xs with
    Nil → Cons x Nil
    Cons y ys →
      if x ≤ y
        then Cons x xs
        else Cons y (insert x ys)
```

example: insertion sort

```
sort :: ???
sort xs =
  match xs with
    Nil → Nil
    Cons y ys →
      insert y (sort ys)
```

```
(≤) :: x:a → y:{a | | 1} → Bool
```

```
insert :: a → List {a| |1} → List a
insert x xs =
  match xs with
    Nil → Cons x Nil
    Cons y ys →
      if x ≤ y
        then Cons x xs
        else Cons y (insert x ys)
```

example: insertion sort

```
sort :: ???
sort xs =
  match xs with
    Nil → Nil
    Cons y ys →
      insert y (sort ys)
```

quadratic!

```
(≤) :: x:a  →  y:{a | | 1}  →  Bool
```

```
insert :: a → List {a| |1} → List a
insert x xs =
  match xs with
    Nil → Cons x Nil
    Cons y ys →
      if x ≤ y
        then Cons x xs
        else Cons y (insert x ys)
```

example: insertion sort

```
sort :: ???
sort xs =
  match xs with
    Nil → Nil
    Cons y ys →
      insert y (sort ys)
```

quadratic!

```
(≤) :: x:a  →  y:{a | | 1}  →  Bool
```

```
insert :: a → List {a| |1} → List a
insert x xs =
  match xs with
    Nil → Cons x Nil
    Cons y ys →
      if x ≤ y
        then Cons x xs
        else Cons y (insert x ys)
```

how can we assign more potential
to the tail of list than the head?

inductive potentials

```
data List a  where
  Nil :: List a
  Cons :: a →
          List a →
          List a
```

inductive potentials

```
data List a where
  Nil :: List a
  Cons :: a →
          List a →
          List a
```

idea: annotate the datatype with distribution of potential!

inductive potentials

```
data QList a where
  Nil :: QList a
  Cons :: a →
          QList a →
          QList a
```

idea: annotate the datatype
with distribution of potential!

inductive potentials

```
data QList a where
  Nil :: QList a
  Cons :: a →
          QList a →
          QList a
```

idea: annotate the datatype with distribution of potential!

inductive potentials

```
data QList a where
  Nil :: QList a
  Cons :: a →
          QList {a | | 1} →
          QList a
```

idea: annotate the datatype with distribution of potential!

inductive potentials

```
data QList a where
  Nil :: QList a
  Cons :: a →
          QList {a | | 1} →
          QList a
```

idea: annotate the datatype with distribution of potential!

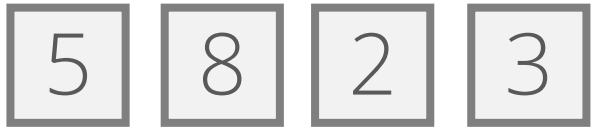
← one more unit of potential than **a** has

inductive potentials

```
data QList a where
  Nil :: QList a
  Cons :: a →
          QList {a | | 1} →
          QList a
```

idea: annotate the datatype with distribution of potential!

one more unit of potential than `a` has



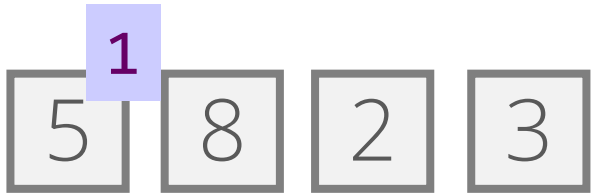
`:: QList {Int | | 1}`

inductive potentials

```
data QList a where
  Nil :: QList a
  Cons :: a →
          QList {a | | 1} →
          QList a
```

idea: annotate the datatype with distribution of potential!

one more unit of potential than `a` has



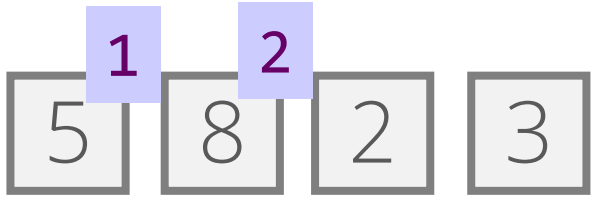
`:: QList {Int | | 1}`

inductive potentials

```
data QList a where
  Nil :: QList a
  Cons :: a →
          QList {a | | 1} →
          QList a
```

idea: annotate the datatype with distribution of potential!

one more unit of potential than `a` has



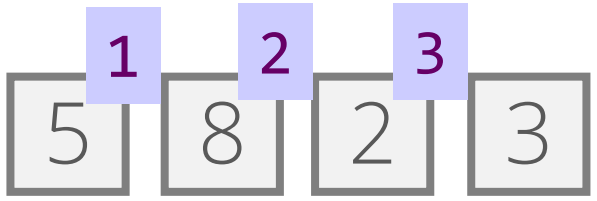
`:: QList {Int | | 1}`

inductive potentials

```
data QList a where
  Nil :: QList a
  Cons :: a →
          QList {a | | 1} →
          QList a
```

idea: annotate the datatype with distribution of potential!

one more unit of potential than `a` has



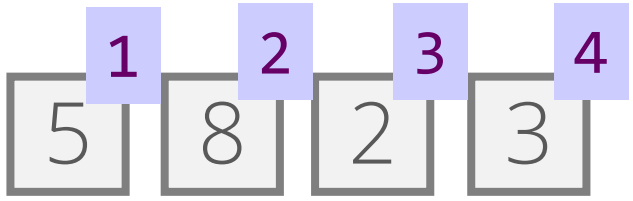
`:: QList {Int | | 1}`

inductive potentials

```
data QList a where
  Nil :: QList a
  Cons :: a →
          QList {a | | 1} →
          QList a
```

idea: annotate the datatype with distribution of potential!

one more unit of potential than `a` has



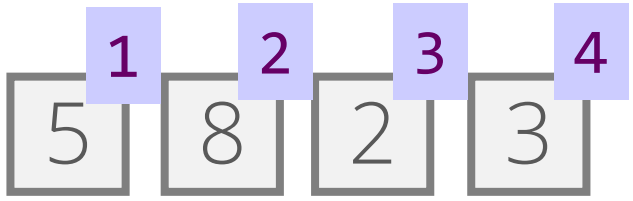
`:: QList {Int | | 1}`

inductive potentials

```
data QList a where
  Nil :: QList a
  Cons :: a →
          QList {a | | 1} →
          QList a
```

idea: annotate the datatype with distribution of potential!

one more unit of potential than `a` has

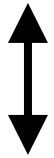


`:: QList {Int | | 1}`

$$\frac{n(n+1)}{2}$$

checking sort

```
sort :: QList {a | | 1} → List a
```



```
sort xs =
  match xs with
    Nil → Nil
    Cons y ys →
      insert y
        (sort ys)
```

checking sort

```
List a
```

```
sort xs =
  match xs with
    Nil → Nil
    Cons y ys →
      insert y
        (sort ys)
```

Context:

```
insert: b → List {b||1} → List b
sort: QList {c||1} → List c
xs: QList {a||1}
```

checking sort

```
List a
```

```
sort xs =
  match xs :: QList {a||1} with
    Nil → Nil
    Cons y ys →
      insert y
        (sort ys)
```

Context:

```
insert: b → List {b||1} → List b
sort: QList {c||1} → List c
xs: QList {a||1}
```

checking sort

```
List a
```

```
sort xs =
  match xs :: QList {a||1} with
    Nil → Nil
    Cons y ys →
      insert y
        (sort ys)
```

Context:

```
insert: b → List {b||1} → List b
sort: QList {c||1} → List c
xs: QList {a||1}
```

```
data QList t where
  Nil :: QList t
  Cons :: t → QList {t||1} → QList t
```

checking sort

```
List a
```

```
sort xs =
  match xs :: QList {a||1} with
    Nil → Nil
    Cons y ys →
      insert y
        (sort ys)
```

Context:

```
insert: b → List {b||1} → List b
sort: QList {c||1} → List c
xs: QList {a||1}
```

`t = {a||1}` so destructing yields

`y: {a||1}` `ys: QList {a||2}`

```
data QList t where
  Nil :: QList t
  Cons :: t → QList {t||1} → QList t
```

checking sort

```
List a
```

```
sort xs =
  match xs with
    Nil → Nil
    Cons y ys →
      insert y
        (sort ys)
```

Context:

```
insert: b → List {b||1} → List b
sort: QList {c||1} → List c
ys: QList {a||2}
y: {a||1}
```

checking sort

```
List a
```

```
sort xs =
  match xs with
    Nil → Nil
    Cons y ys →
      insert y
        (sort (ys::QList{a||2}))
```

Context:

```
insert: b → List {b||1} → List b
sort: QList {c||1} → List c
ys: QList {a||2}
y: {a||1}
```

checking sort

```
List a
```

```
sort xs =
  match xs with
    Nil → Nil
    Cons y ys →
      insert y
        ((sort::QList{a||2} → List {a||1}) (ys::QList{a||2}))
```

polymorphic recursion with
`c = {a||1}`

Context:

```
insert: b → List {b||1} → List b
sort: QList {c||1} → List c
ys: QList {a||2}
y: {a||1}
```

checking sort

```
List a
```

```
sort xs =
  match xs with
    Nil → Nil
    Cons y ys →
      insert y
        ((sort ys) :: List {a||1})
```

polymorphic recursion with `c = {a||1}`

Context:

```
insert: b → List {b||1} → List b
sort: QList {c||1} → List c
ys: QList {a||2}
y: {a||1}
```

checking sort

```
List a
```

```
sort xs =
  match xs with
    Nil → Nil
    Cons y ys →
      insert y
        ((sort ys) :: List {a||1})
```

Context:

```
insert: b → List {b||1} → List b
sort: QList {c||1} → List c
ys: QList {a||0}
y: {a||1}
```

enough linear potential to pay for `insert` with `b = a`

quadratic insertion sort

```
sort :: QList {a| |1} → List a
sort xs =
  match xs with
    Nil → Nil
    Cons y ys →
      insert y (sort ys)
```

```
insert :: a → List {a| |1} → List a
insert x xs =
  match xs with
    Nil → Cons x Nil
    Cons y ys →
      if x ≤ y
        then Cons x xs
        else Cons y (insert x ys)
```

http://comcom.csail.mit.edu/demos/#insertion_sort_coarse

this talk

1. liquid types + resource bounds
2. type checking
3. non-linear bounds
4. value-dependent bounds

insertion sort: fine-grained bound

```
sort :: QList {a| |1} → List a
sort xs =
  match xs with
    Nil → Nil
    Cons y ys →
      insert y (sort ys)
```

```
insert :: a → List {a| |1} → List a
insert x xs =
  match xs with
    Nil → Cons x Nil
    Cons y ys →
      if x ≤ y
        then Cons x xs
        else Cons y (insert x ys)
```

insertion sort: fine-grained bound

```
sort :: QList {a| |1} → List a
sort xs =
  match xs with
    Nil → Nil
    Cons y ys →
      insert y (sort ys)
```

```
insert :: a → List {a| |1} → List a
insert x xs =
  match xs with
    Nil → Cons x Nil
    Cons y ys →
      if x ≤ y
        then Cons x xs
        else Cons y (insert x ys)
```

makes one comparison
per element < x

insertion sort: fine-grained bound

```
sort :: QList {a| |1} → List a
sort xs =
  match xs with
    Nil → Nil
    Cons y ys →
      insert y (sort ys)
```

makes one comparison per decreasing pair of elements

```
insert :: a → List {a| |1} → List a
insert x xs =
  match xs with
    Nil → Cons x Nil
    Cons y ys →
      if x ≤ y
        then Cons x xs
        else Cons y (insert x ys)
```

makes one comparison per element < x

insertion sort: fine-grained bound

```
sort :: ???
sort xs =
  match xs with
    Nil → Nil
    Cons y ys →
      insert y (sort ys)
```

makes one comparison per decreasing pair of elements

```
insert :: ???
insert x xs =
  match xs with
    Nil → Cons x Nil
    Cons y ys →
      if x ≤ y
        then Cons x xs
        else Cons y (insert x ys)
```

makes one comparison per element < x

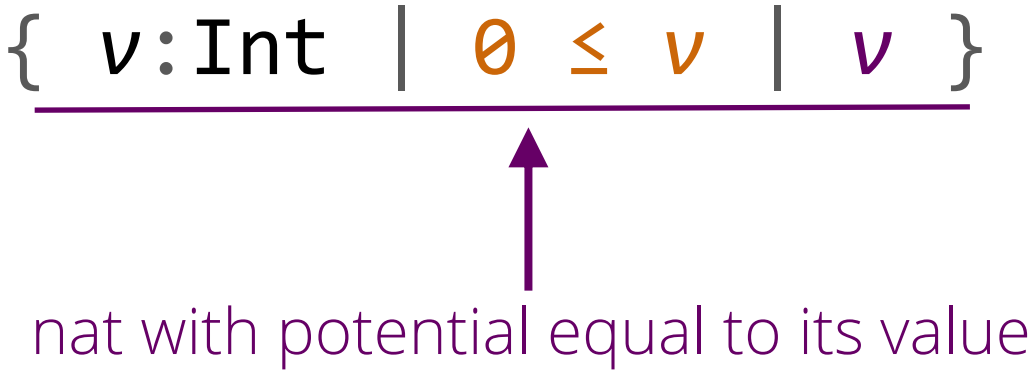
value-dependent potential

$$\{\ \nu{:}\mathtt{Int} \mid 0 \le \nu \mid 1\ \}$$

value-dependent potential

$$\{\ \nu{:}\mathtt{Int} \mid 0 \le \nu \mid \nu\ \}$$

value-dependent potential



insert: fine-grained bound

```
insert ::    a    →   List {a | | 1}  →  List a

insert x xs =
  match xs with
    Nil → Cons x Nil
    Cons y ys →
      if x ≤ y
        then Cons x xs
        else Cons y (insert x ys)
```

makes one comparison per element < x

insert: fine-grained bound

```
insert :: {a||1}  →  List {a||ite (ν < x) 1 0}  →  List a

insert x xs =
  match xs with
    Nil → Cons x Nil
    Cons y ys →
      if x ≤ y
        then Cons x xs
        else Cons y (insert x ys)
```

insert: fine-grained bound

```
insert :: {a||1}  →  List {a||ite (ν < x) 1 0}  →  List a

insert x xs =
  match xs with
    Nil → Cons x Nil
    Cons y ys →
      if x ≤ y
        then Cons x xs
        else Cons y (insert x ys)
```

only assign potential to elements < x

checking insert

```
List a
```

```
insert x xs =
  match xs with
    Nil → Cons x Nil
    Cons y ys →
      if x ≤ y
        then Cons x xs
        else Cons y (insert x ys)
```

Context:

```
insert:{a||1}
  → List {a||ite (v < x) 1 0} → List a
x: {a||1}
y: {a||ite (v < x) 1 0}
ys: List {a||ite (v < x) 1 0}
```

checking insert

```
List a
```

```
insert x xs =
  match xs with
    Nil → Cons x Nil
    Cons y ys →
      if x ≤ y
        then Cons x xs
        else Cons y (insert x ys)
```

Context:

```
insert:{a||1}
  → List {a||ite (ν < x) 1 0} → List a
x: {a||1}
y: {a||ite (ν < x) 1 0}
ys: List {a||ite (ν < x) 1 0}
```

pays for comparison

checking insert

```
List a
```

```
insert x xs =
  match xs with
    Nil → Cons x Nil
    Cons y ys →
      if x ≤ y
        then Cons x xs
        else Cons y (insert (x::{a||1}) ys)
```

Context:

```
insert:{a||1}
  → List {a||ite (ν < x) 1 0} → List a
x: {a||0}
y: {a||ite (ν < x) 1 0}
ys: List {a||ite (ν < x) 1 0}
```

checking insert

```
List a
```

```
insert x xs =
  match xs with
    Nil → Cons x Nil
    Cons y ys →
      if x ≤ y
        then Cons x xs
        else Cons y (insert (x::{a||1}) ys)
```

Context:

```
insert:{a||1}
  → List {a||ite (v < x) 1 0} → List a
x: {a||0}
y: {a||ite (v < x) 1 0}
ys: List {a||ite (v < x) 1 0}
y < x
```

equivalent to {a||1}
in this branch

insertion sort: fine-grained bound

```
sort :: ???
sort xs =
  match xs with
    Nil → Nil
    Cons y ys →
      insert y (sort ys)
```

```
insert :: {a||1}  →
  List {a||ite (ν < x) 1 0} → List a
insert x xs =
  match xs with
    Nil → Cons x Nil
    Cons y ys →
      if x ≤ y
        then Cons x xs
        else Cons y (insert x ys)
```

insertion sort: fine-grained bound

```
sort :: ???
sort xs =
  match xs with
    Nil → Nil
    Cons y ys →
      insert y (sort ys)
```

```
insert :: {a||1}  →
  List {a||ite (ν < x) 1 0} → List a
insert x xs =
  match xs with
    Nil → Cons x Nil
    Cons y ys →
      if x ≤ y
        then Cons x xs
        else Cons y (insert x ys)
```

sort: fine-grained bound

```
sort :: ISList {a||1}  →  List a

sort xs =
  match xs with
    Nil → Nil
    Cons y ys →
      insert y (sort ys)
```

sort: fine-grained bound

```
sort :: ISList {a||1}  →  List a

sort xs =
  match xs with
    Nil → Nil
    Cons y ys →
      insert y (sort ys)
```

```
data ISList a where
  Nil :: ISList a
  Cons :: x:a →
          ISList {a||ite (ν < x) 1 0} →
          ISList t
```

sort: fine-grained bound

```
sort :: ISList {a||1}  →  List a

sort xs =
  match xs with
    Nil → Nil
    Cons y ys →
      insert y (sort ys)
```

```
data ISList a where
  Nil :: ISList a
  Cons :: x:a →
          IS  {a||ite (ν < x) 1 0} →
          ist t
```

not reusable

abstract potentials

```
data List a where
  Nil :: List a
  Cons :: h:a →
          t:List a →
          List a
```

abstract potentials

```
data List a where
  Nil :: List a
  Cons :: h:a →
          t:List a →
          List a
```

idea: parameterize the datatype
by a potential function!

abstract potentials

```
data List a <q: a → a → Int> where
  Nil :: List a
  Cons :: h:a →
          t:List a →
          List a
```

idea: parameterize the datatype
by a potential function!

abstract potentials

```
data List a <q: a → a → Int> where
  Nil :: List a <q>
  Cons :: h:a →
          t:List {a||q h v} <q> →
          List a <q>
```

idea: parameterize the datatype
by a potential function!

insertion sort: fine-grained bound

```
sort xs =
  match xs with
    Nil → Nil
    Cons y ys →
      insert y (sort ys)
```

```
insert x xs =
  match xs with
    Nil → Cons x Nil
    Cons y ys →
      if x ≤ y
        then Cons x xs
        else Cons y (insert x ys)
```

insertion sort: fine-grained bound

```
sort xs =
  match xs with
    Nil → Nil
    Cons y ys →
      insert y (sort ys)
```

```
insert :: {a||1}
  →  List {a||ite (ν < x) 1 0} <0>
  →  List a <0>
insert x xs =
  match xs with
    Nil → Cons x Nil
    Cons y ys →
      if x ≤ y
        then Cons x xs
        else Cons y (insert x ys)
```

insertion sort: fine-grained bound

```
sort ::
  List a <\x y.ite (x < y) 1 0>
  → List a <0>
sort xs =
  match xs with
    Nil → Nil
    Cons y ys →
      insert y (sort ys)
```

```
insert :: {a||1}
  → List {a||ite (v < x) 1 0} <0>
  → List a <0>
insert x xs =
  match xs with
    Nil → Cons x Nil
    Cons y ys →
      if x ≤ y
        then Cons x xs
        else Cons y (insert x ys)
```

insertion sort: fine-grained bound

```
sort ::
  List a <\x y.ite (x < y) 1 0>
  →  List a <0>
sort xs =
  match xs with
    Nil → Nil
    Cons y ys →
      insert y (sort ys)
```

```
insert :: {a||1}
  →  List {a||ite (v < x) 1 0} <0>
  →  List a <0>
insert x xs =
  match xs with
    Nil → Cons x Nil
    Cons y ys →
      if x ≤ y
        then Cons x xs
        else Cons y (insert x ys)
```

http://comcom.csail.mit.edu/demos/#insertion_sort

liquid resource types

1. liquid types + resource bounds
2. type checking

[PLDI'19]

3. non-linear bounds

[ICFP'20 ?]

4. value-dependent bounds