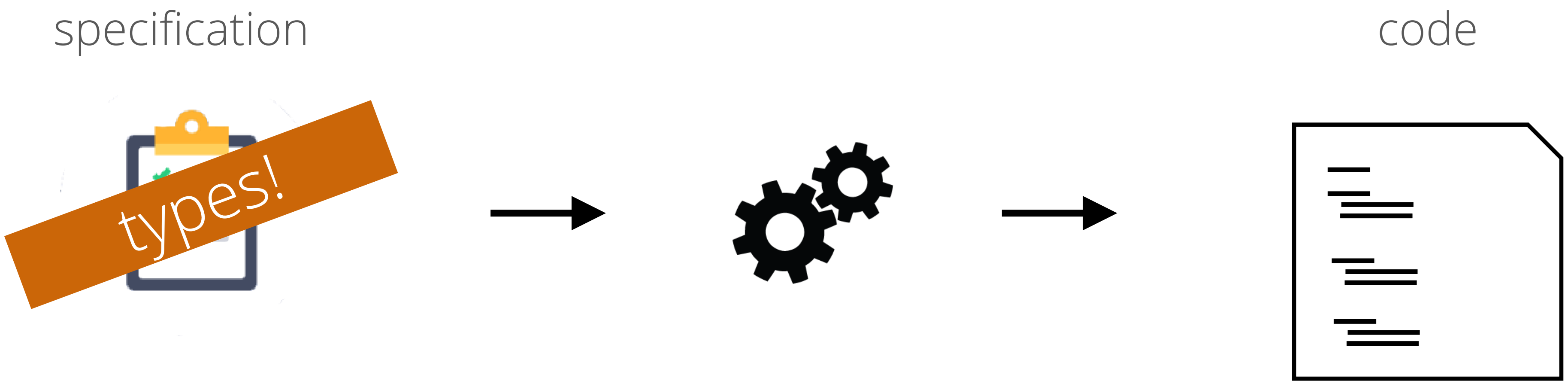
liquid resource types for verification and synthesis

Nadia Polikarpova

with Tristan Knoth, Di Wang, and Jan Hoffmann

UCSDCSE
Computer Science and Engineering

program synthesis

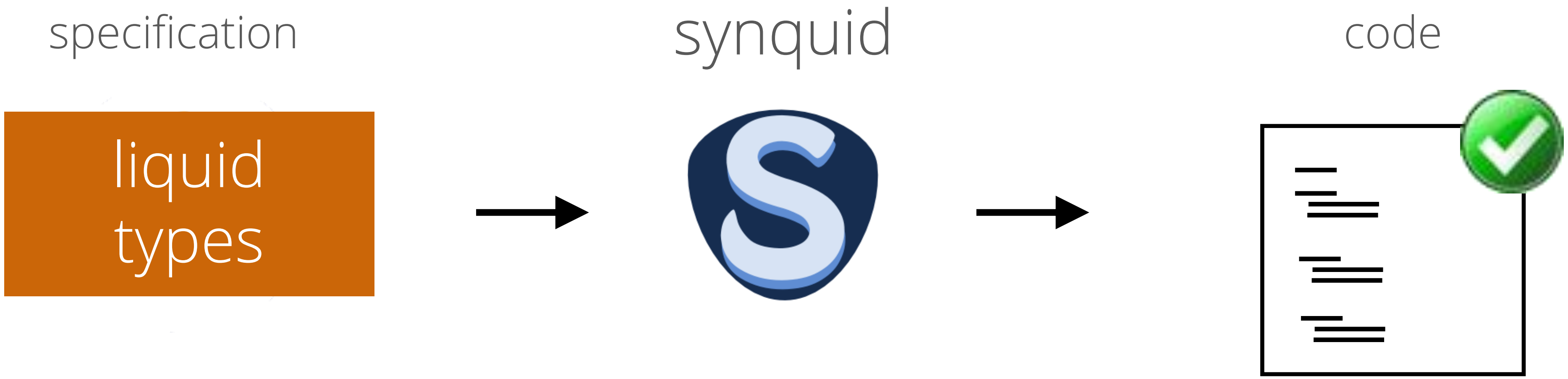


specification

types!

code

type-driven program synthesis



specification

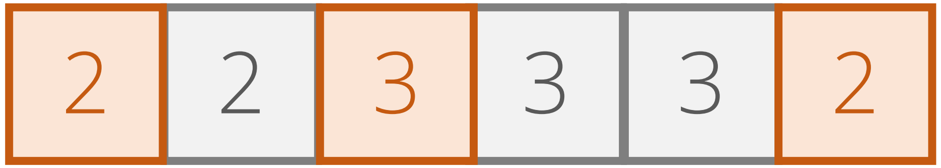
liquid types

synquid

code

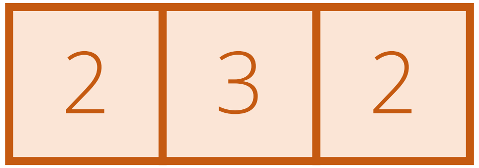
example: compress a list

Input:



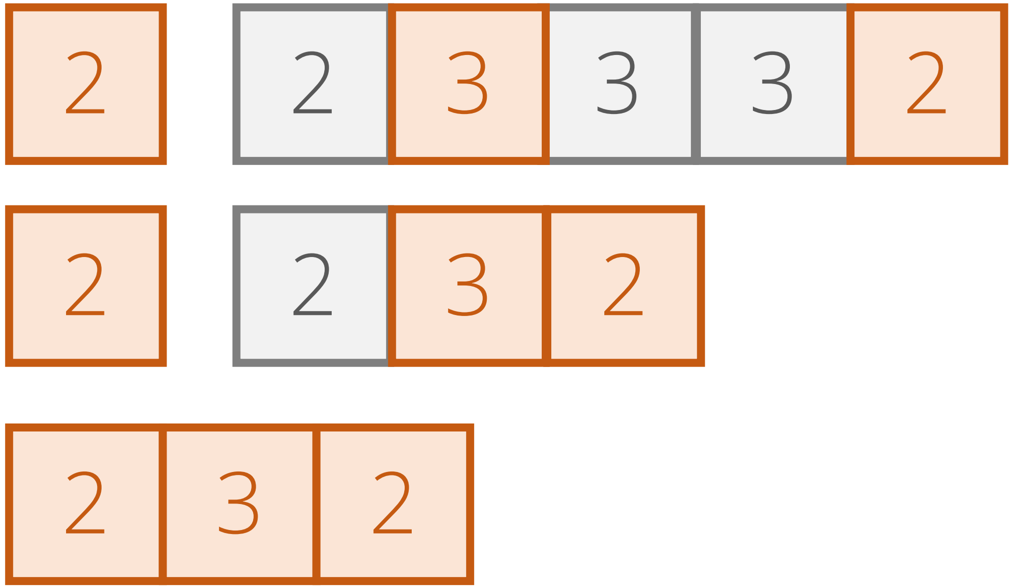
2	2	3	3	3	2

Output:



2	3	2

example: compress a list



synthesizing compress

specification

?

code

?

compress: specification

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

compress: specification

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

compress: generated solution

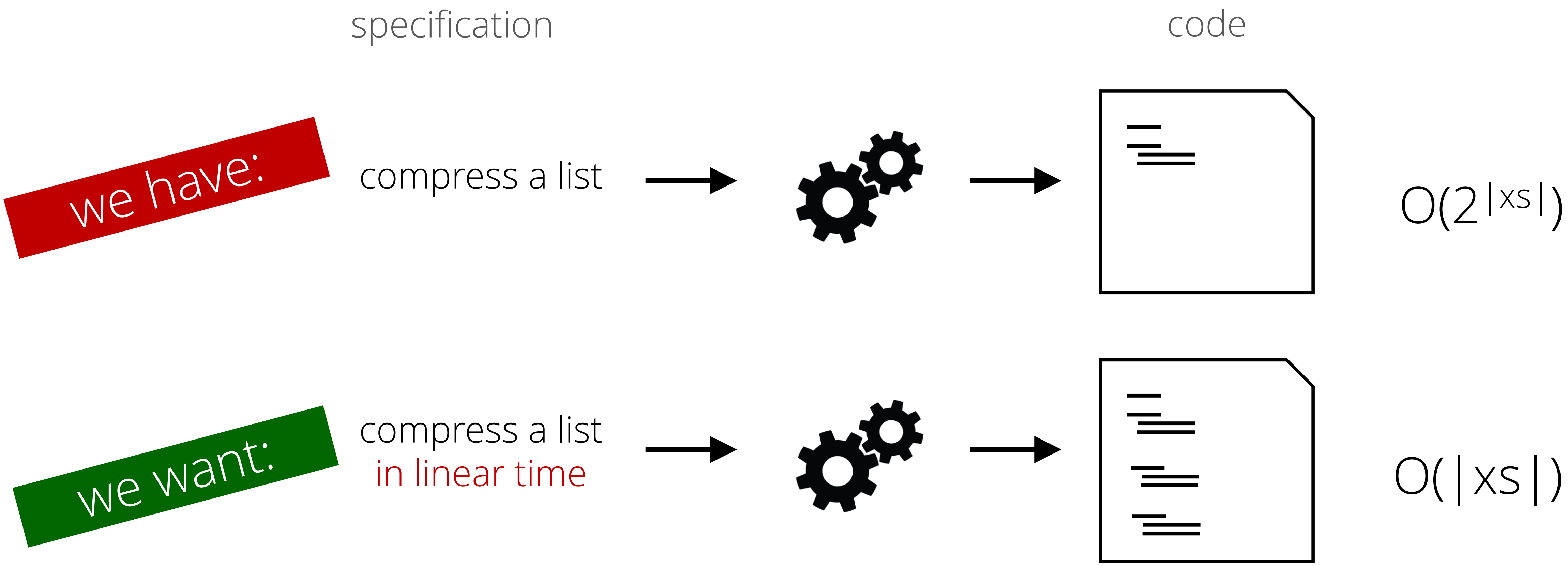
```
compress xs =
  match xs with
    Nil → Nil
    Cons y ys →
      match compress ys with
        Nil → Cons y Nil
        Cons z zs → if y == z
                      then compress ys
                      else Cons y (Cons z zs)
```

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 y == z      Cons z zs
                      then ~~compress ys~~
                      else Cons y (Cons z zs)
```

exponential!

synthesizing efficient programs



specification

code

we have:

compress a list

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

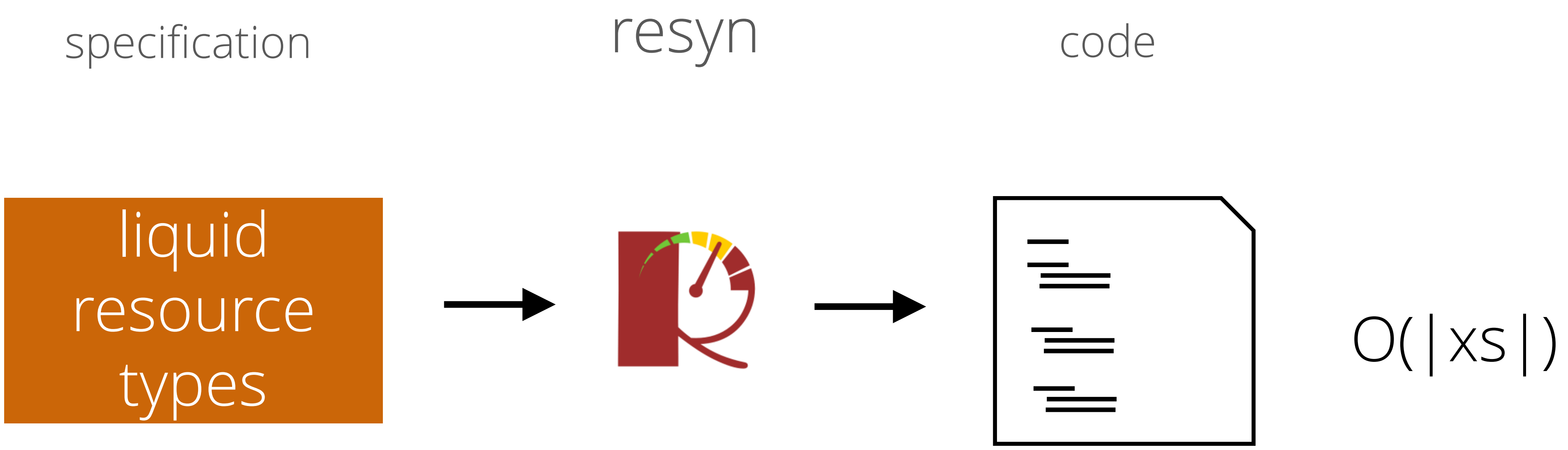
we want:

compress a list

in linear time

$O(|xs|)$

synthesizing efficient programs



specification

resyn

code

liquid resource types

O(|xs|)

this talk

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

[PLDI'19]

under review

this talk

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

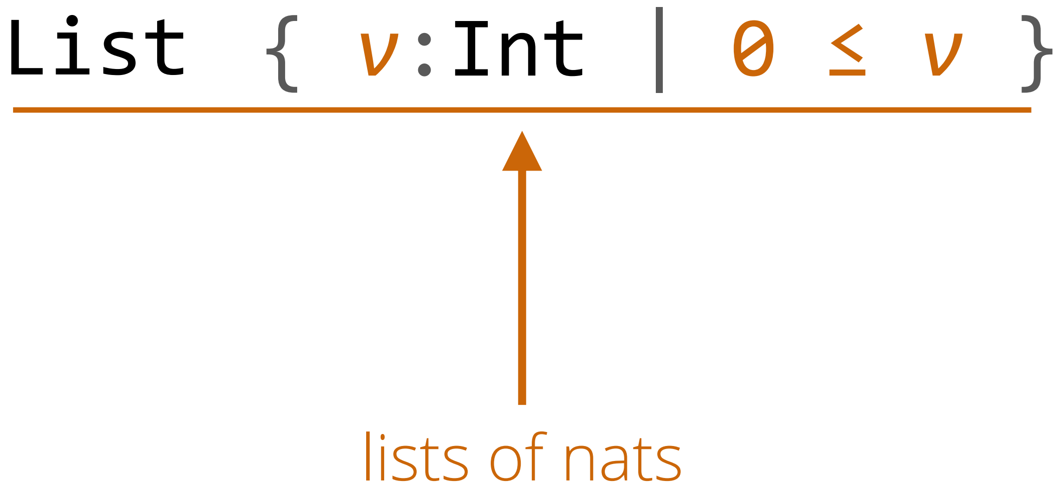
types + refinements

$$\{\ \nu{:}\mathtt{Int} \mid \underbrace{0 \le \nu}_{\text{refinement}}\ \}$$

types + refinements

$$\underbrace{\{\ \nu\mathtt{:Int}\ \mid\ 0 \leq \nu\ \}}_{\text{natural numbers}}$$

types + refinements



```
List { ν:Int | 0 ≤ ν }
```

lists of nats

types + refinements + resources

$$\{\ \nu{:}\mathtt{Int} \mid \underbrace{0 \le \nu}_{\text{refinement}} \mid \underbrace{1}_{\text{potential}}\ \}$$

types + refinements + resources

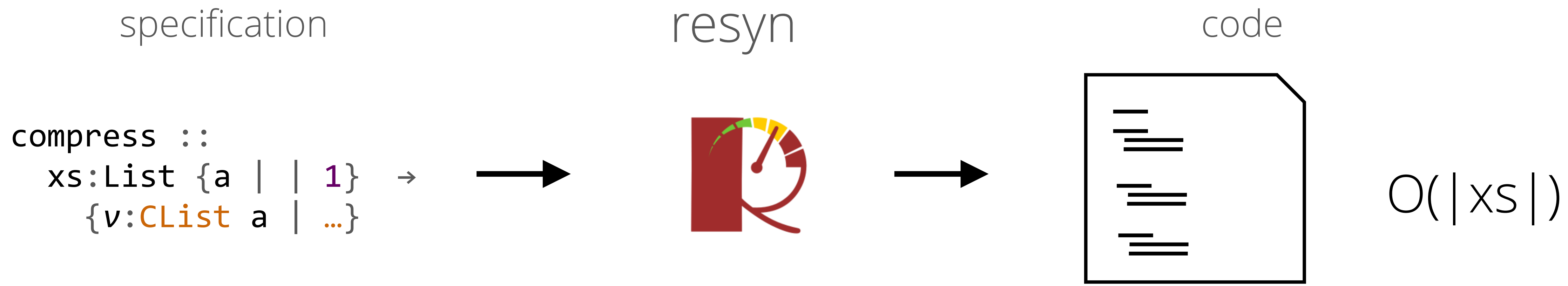
```
List { ν:Int | 0 ≤ ν | 1 }
```

lists of nats with length units of potential

compress: liquid resource type

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

synthesizing linear compress



specification

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

resyn

code

O(|xs|)

this talk

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

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 y == z
                      then compress ys
                      else …
```

checking compress (exponential)

```
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 y == z
                      then compress ys
                      else …
```

use available resources
to pay for recursive calls
and 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 y == z
                      then compress ys
                      else …
```

```
compress: List {a||1} → List a
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 y == z
                    then compress ys
                    else …
```

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

checking compress (exponential)

```
List a
```

↕

```
compress: List {a||1} → List a
ys: List {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 y == z
                      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 y == z
                      then compress (ys :: List {a||q})
                      else …
```

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

1. total potential must be partitioned into two uses:

Constraints: ∃p,q:

$$1 = p + q$$

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 y == z
                      then compress (ys :: List {a||q})
                      else …
```

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

2. *p* must be enough to call compress

Constraints: ∃*p*,*q*:

$$1 = p + q$$

$$p \geq 1$$

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 y == z
                    then compress (ys :: List {a||q})
                    else …
```

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

3. q must be enough to call compress

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

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

Constraints: ∃p,q:

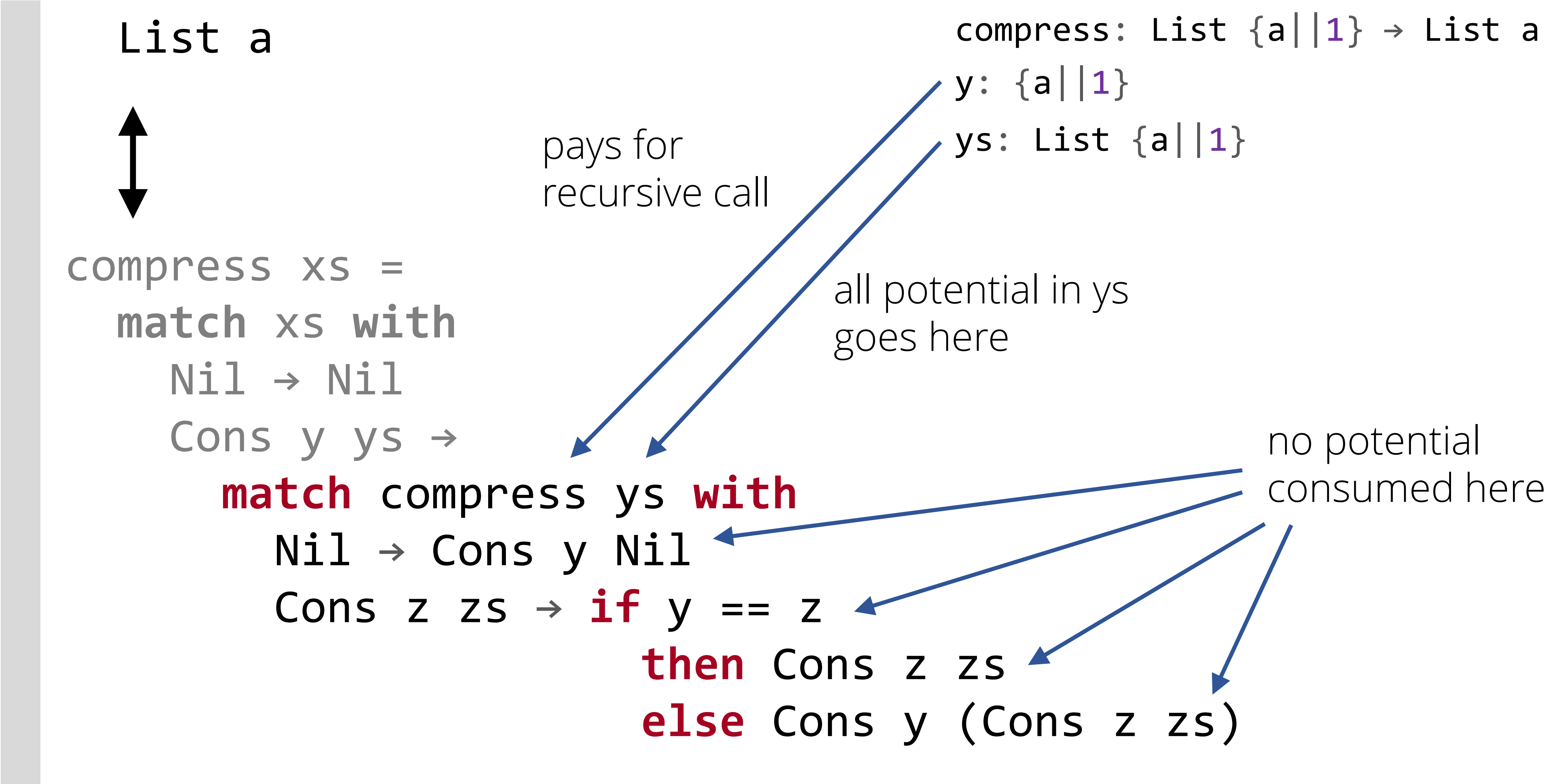
$$1 = p + q$$

$$p \geq 1$$

$$q \geq 1$$

SMT solver: UNSAT!

checking compress (linear)



```
  List a

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

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

pays for
recursive call

all potential in ys
goes here

no potential
consumed here

subtyping

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

sharing

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

resource polymorphism for free

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


compress2 :: List {b | | 2}  →  List b
compress2 xs = compress (compress xs)
```

resource polymorphism for free

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


compress2 :: List {b | | 2}  →  List b
compress2 xs = compress[b / a] (compress[{b||1} / a] xs)
```

this talk

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

value-dependent potential

$$\underbrace{\{\ \nu{:}\mathtt{Int}\ |\ 0 \le \nu\ |\ \nu\ \}}_{\text{nat with potential equal to its value}}$$

insert into sorted list

```
insert :: x:a  →  xs:SList {a | | 1}
          →  SList 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 step
per element < x?

insert: dependent bound

```
insert :: x:a  →  xs:SList {a | | ν < x ? 1 : 0}
             →  SList 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)
```

y: {a|| ν < x ? 1 : 0}

pays for
recursive call

this talk

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

insertion sort

```
insert :: x:a
  → xs:SList {a| |ν < x ? 1 : 0}
  → SList 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 step
per element < x

```
sort :: ???
```

super-linear!

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

makes one step per out-of-order pair of elements

sorted list via inductive refinements

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

sorted list via inductive refinements

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

sorted list via inductive refinements

```
data SList a where
  Nil :: SList a
  Cons :: h:a →
          t:SList {a | h ≤ ν} →
          SList a
```

quadratic list via inductive potentials

```
data SList a where
  Nil :: SList a
  Cons :: h:a →
    t:SList {a | h ≤ ν} →
    SList a
```

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

generality via abstract refinements

```
data SList a where
  Nil :: SList a
  Cons :: h:a →
    t:SList {a | h ≤ ν} →
    SList a
```

```
data List a <p: a → a → Bool> where
  Nil :: List a <p>
  Cons :: h:a →
    t: List {a | p h ν} <p> →
    List a <p>

type SList a = List a < _0 ≤ _1 >
```

generality via abstract potentials

```
data List a
       <p: a → a → Bool> where
  Nil :: List a <p>
  Cons :: h:a →
    t: List {a | p h ν} <p> →
    List a <p>

type SList a
   = List a < _0 ≤ _1 >
```

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

type QList a
   = List a < _1 < _0 ? 1 : 0 >
```

insertion sort

```
insert :: a
  → SList {a| |ν < x ? 1 : 0}
  → SList 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 :: List a <_1 < _0 ? 1 : 0 >
  → SList a
```

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

liquid resource types

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

+ talk to me about:

- “logarithmic” bounds via trees
- linear types for program synthesis