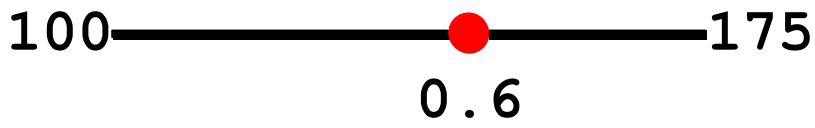
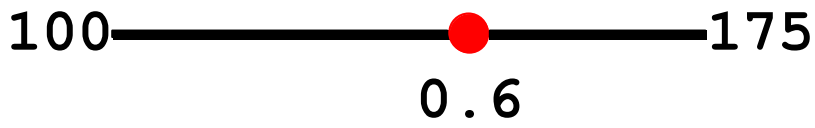
Typing Rank Polymorphism

Justin Slepak

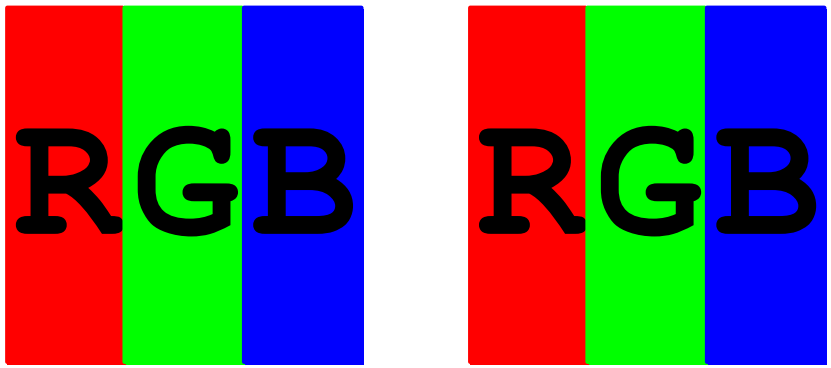
In collaboration with Olin Shivers and Panagiotis Manolios



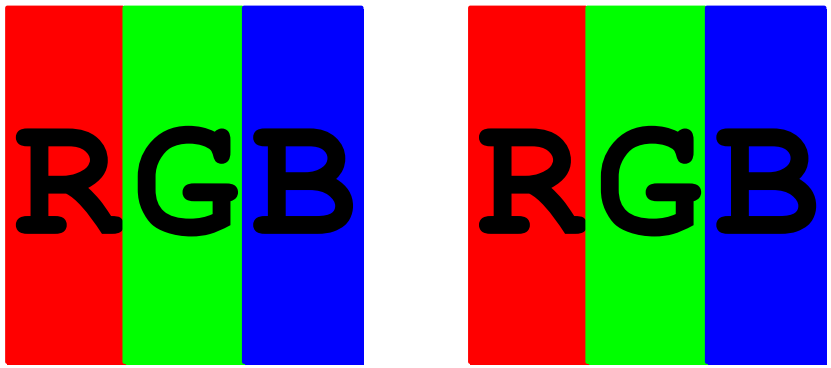

```
(λ [(lo 0) (hi 0) (α 0)]
  (+ (* hi α) (* lo (- 1 α))))
```



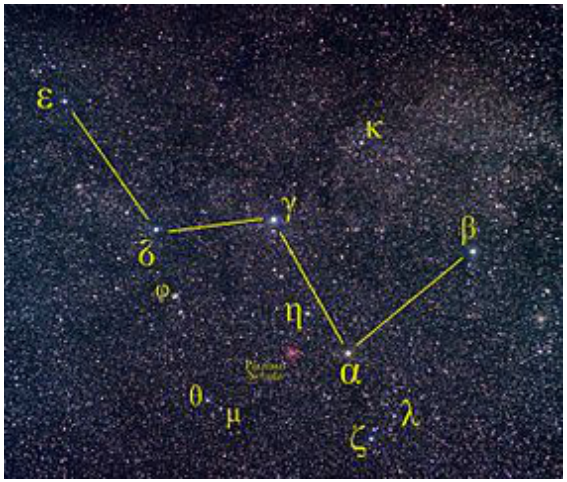

```
((λ [(lo 0) (hi 0) (α 0)]
   (+ (* hi α) (* lo (- 1 α))))
 100
 175
 0.6)
```




```
(λ [(lo 0) (hi 0) (α 0)]
  (+ (* hi α) (* lo (- 1 α))))
```

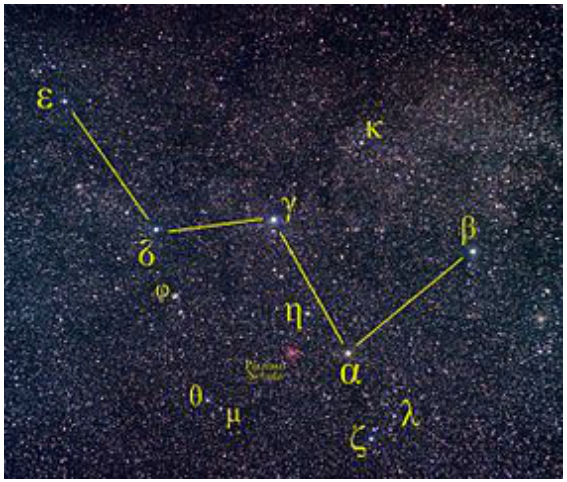



```
((λ [(lo 0) (hi 0) (α 0)]
   (+ (* hi α) (* lo (- 1 α))))
 rgb1 ; 3 channels
 rgb2 ; 3 channels
 0.6) ; scalar
```



Credit: Wikimedia user Sadalsuud

```
(λ [(lo 0) (hi 0) (α 0)]
  (+ (* hi α) (* lo (- 1 α))))
```



Credit: Wikimedia user Sadalsuud

```
((λ [(lo 0) (hi 0) (α 0)]
   (+ (* hi α) (* lo (- 1 α))))
 sky         ; row × col × chan
 labels      ; row × col × chan
 img-mask)   ; row × col × chan
```

```
(λ [(lo 0) (hi 0) (α 0)]
  (+ (* hi α) (* lo (- 1 α))))
```

```
((λ [(lo 0) (hi 0) (α 0)]
   (+ (* hi α) (* lo (- 1 α))))
 film       ; time × row × col × chan
 audience   ; time × row × col × chan
 vid-mask)  ; time × row × col × chan
```

Credit: Wikimedia user Thetawave

```
(λ [(lo 0) (hi 0) (α 0)]
  (+ (* hi α) (* lo (- 1 α))))
```

Credit: Wikimedia user Thetawave

```
((λ [(lo 0) (hi 0) (α 0)]
   (+ (* hi α) (* lo (- 1 α))))
 scene1     ; time × row × col × chan
 scene2     ; time × row × col × chan
 [0.0 ... 1.0]) ; time
```

```
(lerp
 dragonfly   ; row × col × chan
 tint-color  ; chan
 0.6)        ; scalar
```

```
(~[1 1 0]lerp
 dragonfly   ; row × col × chan
 tint-color  ; chan
 0.6)        ; scalar
```

```
(~[1 1 0]lerp
 dragonfly   ; row × col × chan
 tint-color  ; chan
 0.6)        ; scalar
```

```
(~[1 1 0]lerp
 dragonfly   ; row × col × chan
 tint-color  ; chan
 0.6)        ; scalar
```

```
(~[1 1 0]lerp
 dragonfly   ; row × col × chan
 tint-color  ; chan
 0.6)        ; scalar
```

```
(~[1 1 0]lerp
 dragonfly   ; row × col × chan
 tint-color  ; chan
 0.6)        ; scalar
```

```
(~[1 1 0]lerp
 dragonfly   ; row × col × chan
 tint-color  ; chan
 0.6)        ; scalar
```

```
(~[1 1 0]lerp
 dragonfly   ; row × col × chan
 tint-color  ; chan
 0.6)        ; scalar
```

```
(~[1 1 0]lerp
 dragonfly   ; row × col × chan
 tint-color  ; chan
 0.6)        ; scalar
```

Rank manipulation

```
(~[0 1]* [1 2 3] [10 20])
```

```
(~[0 1]* [1 2 3] [10 20])
```

Cells:

```
(~[0 1]* [1 2 3] [10 20])
```

Cells: `~[0 1]*`

```
(~[0 1]* [1 2 3] [10 20])
```

Cells:

```
~[0 1]*  1
         2
         3
```

```
(~[0 1]* [1 2 3] [10 20])
```

Cells:

```
~[0 1]*
```

```
1
2
3
```

```
[10 20]
```

```
(~[0 1]* [1 2 3] [10 20])
```

Cells:	~[0 1]*	1 2 3	[10 20]
Frame:	[]	[3]	[]

```
(~[0 1]* [1 2 3] [10 20])
```

Cells:

```
~[0 1]*  1       [10 20]
         2
         3
```

Frame:

```
[]       [3]     []
```

```
[(* 1 [10 20])
 (* 2 [10 20])
 (* 3 [10 20])]
```

```
          (~[0 1]* [1 2 3] [10 20])

Cells:     ~[0 1]*  1       [10 20]
                    2
                    3
Frame:      []      [3]      []

  [(* 1 [10 20])        [[10 20]
   (* 2 [10 20])         [20 40]
   (* 3 [10 20])]        [30 60]]
```

```
; M×N array for M points in N-space
> (define points ...)
```

```
; M×N array for M points in N-space
> (define points ...)

> (define (dist^2 (a 1) (b 1))
    (reduce + 0 (sqr (- a b))))
```

```
; M×N array for M points in N-space
> (define points ...)

> (define (dist^2 (a 1) (b 1))
    (reduce + 0 (sqr (- a b))))

; points[x,newDim,y] and points[newDim,x,y]?
```

```
; M×N array for M points in N-space
> (define points ...)

> (define (dist^2 (a 1) (b 1))
    (reduce + 0 (sqr (- a b))))

; points[x,newDim,y] and points[newDim,x,y]?
> (~[1 2]dist^2 points points)
[[some ...]
 [symmetric ...]
 [matrix ...] ...]
```

Non-rectangular data

```
> (define ragged [(box [1])
                  (box [4 2 5])
                  (box [8 7 2 4 3])
                  (box [9 0])])
```

```
> (define ragged [(box [1])
                  (box [4 2 5])
                  (box [8 7 2 4 3])
                  (box [9 0])])
> (shape-of ragged)
```

```
> (define ragged [(box [1])
                  (box [4 2 5])
                  (box [8 7 2 4 3])
                  (box [9 0])])
> (shape-of ragged)
[4]
```

```
> (define ragged [(box [1])
                  (box [4 2 5])
                  (box [8 7 2 4 3])
                  (box [9 0])])
> (shape-of ragged)
[4]
> ((λ (b 0) (unbox contents b
                 (shape-of contents)))
   ragged)
```

```
> (define ragged [(box [1])
                  (box [4 2 5])
                  (box [8 7 2 4 3])
                  (box [9 0])])
> (shape-of ragged)
[4]
> ((λ (b 0) (unbox contents b
                   (shape-of contents)))
   ragged)
[[1] [3] [5] [2]]
```

```
> (take 3 [1 2 3 4 5])
```

```
> (take 3 [1 2 3 4 5])
[1 2 3]
```

```
> (take 3 [1 2 3 4 5])
[1 2 3]
> (take 2 [1 2 3 4 5])
```

```
> (take 3 [1 2 3 4 5])
[1 2 3]
> (take 2 [1 2 3 4 5])
[1 2]
```

```
> (take 3 [1 2 3 4 5])
[1 2 3]
> (take 2 [1 2 3 4 5])
[1 2]
> (take [2 3] [1 2 3 4 5])
```

```
> (take 3 [1 2 3 4 5])
[1 2 3]
> (take 2 [1 2 3 4 5])
[1 2]
> (take [2 3] [1 2 3 4 5])
Error: Result cells have mismatched shapes
```

```
> (take 3 [1 2 3 4 5])
[1 2 3]
> (take 2 [1 2 3 4 5])
[1 2]
> (take [2 3] [1 2 3 4 5])
Error: Result cells have mismatched shapes
```

Ragged results can arise from lifting non-ragged ops

```
> (take* 3 [1 2 3 4 5])
```

```
> (take* 3 [1 2 3 4 5])
(box [1 2 3])
```

```
> (take* 3 [1 2 3 4 5])
(box [1 2 3])
> (take* [2 4] [1 2 3 4 5])
```

```
> (take* 3 [1 2 3 4 5])
(box [1 2 3])
> (take* [2 4] [1 2 3 4 5])
[(box [1 2])
 (box [1 2 3 4])]
```

```
> (take* 3 [1 2 3 4 5])
(box [1 2 3])
> (take* [2 4] [1 2 3 4 5])
[(box [1 2])
 (box [1 2 3 4])]
> (take* (add1 (iota [5]))
         [1 2 3 4 5])
```

```
> (take* 3 [1 2 3 4 5])
(box [1 2 3])
> (take* [2 4] [1 2 3 4 5])
[(box [1 2])
 (box [1 2 3 4])]
> (take* (add1 (iota [5]))
         [1 2 3 4 5])
[(box [1])
 (box [1 2])
 (box [1 2 3])
 (box [1 2 3 4])
 (box [1 2 3 4 5])]
```

```
> (define (prefixes (v all))
    ; Expect arg with at least one dimension
    (take* (add1 (iota [(length v)])) v))
```

```
> (define (prefixes (v all))
    ; Expect arg with at least one dimension
    (take* (add1 (iota [(length v)])) v))

> (define (multi-dist^2 (src 1) (dst-box 0))
    (unbox dsts dst-box
      (box (dist^2 src dsts))))
```

```
> (define (prefixes (v all))
    ; Expect arg with at least one dimension
    (take* (add1 (iota [(length v)])) v))

> (define (multi-dist^2 (src 1) (dst-box 0))
    (unbox dsts dst-box
      (box (dist^2 src dsts))))

> (multi-dist^2 points (prefixes points))
```

```
> (define (prefixes (v all))
    ; Expect arg with at least one dimension
    (take* (add1 (iota [(length v)])) v))

> (define (multi-dist^2 (src 1) (dst-box 0))
    (unbox dsts dst-box
      (box (dist^2 src dsts))))

> (multi-dist^2 points (prefixes points))
[(box [p1->p1])
 (box [p2->p1 p2->p2])
 (box [p3->p1 p3->p2 p3->p3])
 ...]
```

Heterogeneous records

```
> (define (champion (y 0) (c 0) (n 0))
    {(year y) (city c) (name n)})
```

```
> (define (champion (y 0) (c 0) (n 0))
    {(year y) (city c) (name n)})

> (champion 2018 "Washington" "Capitals")
```

```
> (define (champion (y 0) (c 0) (n 0))
    {(year y) (city c) (name n)})

> (champion 2018 "Washington" "Capitals")
{(year 2018)
 (city "Washington")
 (name "Capitals")}
```

```
> (define (champion (y 0) (c 0) (n 0))
    {(year y) (city c) (name n)})

> (champion 2018 "Washington" "Capitals")
{(year 2018)
 (city "Washington")
 (name "Capitals")}
> (define last-four
    (champion [2015 2016 2017 2018]
              ["Chicago"
               "Pittsburgh"
               "Pittsburgh"
               "Washington"]
              ["Blackhawks"
               "Penguins"
               "Penguins"
               "Capitals"]))
```

```
> last-four
```

```
> last-four
[{(year 2015) (city "Chicago") (name "Blackhawks")}
 {(year 2016) (city "Pittsburgh") (name "Penguins")}
 {(year 2017) (city "Pittsburgh") (name "Penguins")}
 {(year 2018) (city "Washington") (name "Capitals")}]
```

```
> last-four
[{(year 2015) (city "Chicago") (name "Blackhawks")}
 {(year 2016) (city "Pittsburgh") (name "Penguins")}
 {(year 2017) (city "Pittsburgh") (name "Penguins")}
 {(year 2018) (city "Washington") (name "Capitals")}]
> (shape-of last-four)
```

```
> last-four
[{(year 2015) (city "Chicago") (name "Blackhawks")}
 {(year 2016) (city "Pittsburgh") (name "Penguins")}
 {(year 2017) (city "Pittsburgh") (name "Penguins")}
 {(year 2018) (city "Washington") (name "Capitals")}]
> (shape-of last-four)
[4]
```

```
> last-four
[{(year 2015) (city "Chicago") (name "Blackhawks")}
 {(year 2016) (city "Pittsburgh") (name "Penguins")}
 {(year 2017) (city "Pittsburgh") (name "Penguins")}
 {(year 2018) (city "Washington") (name "Capitals")}]
> (shape-of last-four)
[4]
> (head last-four)
```

```
> last-four
[{(year 2015) (city "Chicago") (name "Blackhawks")}
 {(year 2016) (city "Pittsburgh") (name "Penguins")}
 {(year 2017) (city "Pittsburgh") (name "Penguins")}
 {(year 2018) (city "Washington") (name "Capitals")}]
> (shape-of last-four)
[4]
> (head last-four)
{(year 2015)
 (city "Chicago")
 (name "Blackhawks")}
```

```
> last-four
[{(year 2015) (city "Chicago") (name "Blackhawks")}
 {(year 2016) (city "Pittsburgh") (name "Penguins")}
 {(year 2017) (city "Pittsburgh") (name "Penguins")}
 {(year 2018) (city "Washington") (name "Capitals")}]
> (shape-of last-four)
[4]
> (head last-four)
{(year 2015)
 (city "Chicago")
 (name "Blackhawks")}
> (get city last-four)
```

```
> last-four
[{(year 2015) (city "Chicago") (name "Blackhawks")}
 {(year 2016) (city "Pittsburgh") (name "Penguins")}
 {(year 2017) (city "Pittsburgh") (name "Penguins")}
 {(year 2018) (city "Washington") (name "Capitals")}]
> (shape-of last-four)
[4]
> (head last-four)
{(year 2015)
 (city "Chicago")
 (name "Blackhawks")}
> (get city last-four)
["Chicago" "Pittsburgh" "Pittsburgh" "Washington"]
```

```
> last-four
[{(year 2015) (city "Chicago") (name "Blackhawks")}
 {(year 2016) (city "Pittsburgh") (name "Penguins")}
 {(year 2017) (city "Pittsburgh") (name "Penguins")}
 {(year 2018) (city "Washington") (name "Capitals")}]
> (shape-of last-four)
[4]
> (head last-four)
{(year 2015)
 (city "Chicago")
 (name "Blackhawks")}
> (get city last-four)
["Chicago" "Pittsburgh" "Pittsburgh" "Washington"]
> (string=? "Pittsburgh" (get city last-four))
```

```
> last-four
[{(year 2015) (city "Chicago") (name "Blackhawks")}
 {(year 2016) (city "Pittsburgh") (name "Penguins")}
 {(year 2017) (city "Pittsburgh") (name "Penguins")}
 {(year 2018) (city "Washington") (name "Capitals")}]
> (shape-of last-four)
[4]
> (head last-four)
{(year 2015)
 (city "Chicago")
 (name "Blackhawks")}
> (get city last-four)
["Chicago" "Pittsburgh" "Pittsburgh" "Washington"]
> (string=? "Pittsburgh" (get city last-four))
[#f #t #t #f]
```

Formalism

e ::=		*Expressions*
	x	*Variable reference*
|	`(array (`$n\ \cdots$`) `$\mathfrak{a}\ \cdots$`)`	*Array, containing atoms*
|	`(frame (`$n\ \cdots$`) `$e\ \cdots$`)`	*Frame of subarray cells*
|	`(`$e_f\ e_a\ \cdots$`)`	*Term application*
|	`(t-app `$e_f\ \iota_a\ \cdots$`)`	*Type application*
|	`(i-app `$e_f\ \iota_a\ \cdots$`)`	*Index application*
|	`(unbox (`$x_i\ \cdots\ x_e\ e_s$`) `e_b`)`	*Let-binding box contents*
$\mathfrak{a}$ ::=		*Atoms*
	$\mathfrak{b}$	*Base value*
|	$\mathfrak{o}$	*Primitive operator*
|	`(`λ `((`$x\ \tau$`)`$\cdots$`) `e`)`	*Term abstraction*
|	`(T`λ `((`$x\ k$`)`$\cdots$`) `v`)`	*Type abstraction*
|	`(I`λ `((`$x\ \gamma$`)`$\cdots$`) `v`)`	*Index abstraction*
|	`(box `$\iota\ \cdots\ v\ \tau$`)`	*Index abstraction*

τ	$::=$	*Types*
	B	*Base types*
$\mid$	`(->` $(\tau \cdots)$ τ`)`	*Functions*
$\mid$	`(Forall ((`x k`)` $\cdots$ `)` τ`)`	*Universal types*
$\mid$	`(Pi ((`x γ`)` $\cdots$ `)` τ`)`	*Dependent products*
$\mid$	`(Sigma ((`x γ`)` $\cdots$ `)` τ`)`	*Dependent sums*
$\mid$	`(Arr` ι τ`)`	*Arrays*
ι	$::=$	*Type indices*
	x	*Index variable*
$\mid$	n	*Natural number*
$\mid$	`(Shp` $\iota \cdots$ `)`	*Shape*
$\mid$	`(+` $\iota \cdots$ `)`	*Adding naturals*
$\mid$	`(++` $\iota \cdots$ `)`	*Appending shapes*
k	$::=$	*Kinds*
	`Atom`	*Dimension: one natural*
$\mid$	`Array`	*Shape: sequence of naturals*
γ	$::=$	*Index sorts*
	`Dim`	*Dimension: one natural*
$\mid$	`Shape`	*Shape: sequence of naturals*

Machine-friendly, fully-annotated syntax

```
((array () +)(Arr (Shp) (-> ((Arr (Shp) Int)(Arr (Shp) Int)) (Arr (Shp) Int)))
 (array (3) 1 2 3)(Arr (Shp 3) Int)
 (array (3 2) 10 20 30 40 50 60)(Arr (Shp 32) Int))
```

Machine-friendly, fully-annotated syntax

```
((array () +)(Arr (Shp) (-> ((Arr (Shp) Int)(Arr (Shp) Int)) (Arr (Shp) Int)))
 (array (3) 1 2 3)(Arr (Shp 3) Int)
 (array (3 2) 10 20 30 40 50 60)(Arr (Shp 32) Int))
```

Human-friendly shorthand

```
(+ [1 2 3] [[10 20] [30 40] [50 60]])
```

$\Theta; \Delta; \Gamma \vdash t : \tau$ Given index vars Θ,
type vars Δ,
term vars Γ,
term t has type τ.

$$\Theta; \Delta; \Gamma \vdash t : \tau \qquad \text{Given index vars } \Theta, \text{ type vars } \Delta, \text{ term vars } \Gamma, \text{ term } t \text{ has type } \tau.$$

$$\frac{\Theta; \Delta; \Gamma \vdash \mathfrak{a}_j : \tau \text{ for each } j \qquad \Theta; \Delta \vdash \tau :: \texttt{Atom} \qquad \mathit{Length}\,[\![\mathfrak{a}\ \cdots]\!] = \prod n\ \cdots}{\Theta; \Delta; \Gamma \vdash \texttt{(array (}n\ \cdots\texttt{) }\mathfrak{a}\ \cdots\texttt{) : (Arr (Shp }n\ \cdots\texttt{) }\tau\texttt{)}} \quad \text{T-Array}$$

$$\Theta; \Delta; \Gamma \vdash t : \tau$$

Given index vars Θ,
type vars Δ,
term vars Γ,
term t has type τ.

$$\frac{\Theta; \Delta; \Gamma \vdash \mathfrak{a}_j : \tau \text{ for each } j \qquad \Theta; \Delta \vdash \tau :: \texttt{Atom} \qquad \mathit{Length}\,[\![\mathfrak{a}\ \cdots]\!] = \prod n\ \cdots}{\Theta; \Delta; \Gamma \vdash \texttt{(array (}n\ \cdots\texttt{) }\mathfrak{a}\ \cdots\texttt{) : (Arr (Shp }n\ \cdots\texttt{) }\tau\texttt{)}} \quad \text{T-Array}$$

$$\frac{\Theta; \Delta; \Gamma \vdash e_j : \texttt{(Arr }\iota\ \tau\texttt{)} \text{ for each } j \qquad \Theta; \Delta \vdash \texttt{(Arr }\iota\ \tau\texttt{)} :: \texttt{Array} \qquad \mathit{Length}\,[\![e\ \cdots]\!] = \prod n\ \cdots}{\Theta; \Delta; \Gamma \vdash \texttt{(frame (}n\ \cdots\texttt{) }e\ \cdots\texttt{) : (Arr (++ (Shp }n\ \cdots\texttt{) }\iota\texttt{) }\tau\texttt{)}} \quad \text{T-Frame}$$

$$\frac{\begin{array}{c}\Theta;\Delta;\Gamma \vdash e_f : \texttt{(Arr } \iota_f \texttt{ (->} \texttt{ ((Arr } \iota_i\ \tau_i\texttt{) } \cdots \texttt{) (Arr } \iota_o\ \tau_o\texttt{)))} \\ \Theta;\Delta;\Gamma \vdash e_{aj} : \texttt{(Arr (++ } \iota_{aj}\ \iota_{ij}\texttt{) } \tau_i\texttt{)} \ \text{ for each } j \\ \iota_p = \bigsqcup [\![\iota_f\ \iota_a\ \cdots]\!]\end{array}}{\Theta;\Delta;\Gamma \vdash \texttt{(}e_f\ e_a\ \cdots\texttt{)} : \texttt{(Arr (++ } \iota_p\ \iota_o\texttt{) } \tau_o\texttt{)}} \quad \textsc{T-App}$$

$$\frac{\begin{array}{c}\Theta;\Delta;\Gamma \vdash e_f : (\texttt{Arr}\ \iota_f\ (\texttt{->}\ ((\texttt{Arr}\ \iota_i\ \tau_i)\ \cdots)\ (\texttt{Arr}\ \iota_o\ \tau_o))) \\ \Theta;\Delta;\Gamma \vdash e_{aj} : (\texttt{Arr}\ (\texttt{++}\ \iota_{aj}\ \iota_{ij})\ \tau_i)\ \text{ for each } j \\ \iota_p = \bigsqcup [\![\iota_f\ \iota_a\ \cdots]\!]\end{array}}{\Theta;\Delta;\Gamma \vdash (e_f\ e_a\ \cdots) : (\texttt{Arr}\ (\texttt{++}\ \iota_p\ \iota_o)\ \tau_o)}\ \text{T-App}$$

```
(+ [1 2 3] [[10 20] [30 40] [50 60]])
```

$$\frac{\Theta;\Delta;\Gamma \vdash e_f : (\texttt{Arr}\ \iota_f\ (\texttt{->}\ ((\texttt{Arr}\ \iota_i\ \tau_i)\ \cdots)\ (\texttt{Arr}\ \iota_o\ \tau_o))) \quad \Theta;\Delta;\Gamma \vdash e_{aj} : (\texttt{Arr}\ (\texttt{++}\ \iota_{aj}\ \iota_{ij})\ \tau_i)\ \text{ for each } j \quad \iota_p = \bigsqcup [\![\iota_f\ \iota_a\ \cdots]\!]}{\Theta;\Delta;\Gamma \vdash (e_f\ e_a\ \cdots) : (\texttt{Arr}\ (\texttt{++}\ \iota_p\ \iota_o)\ \tau_o)}\ \text{T-App}$$

```
(+ [1 2 3] [[10 20] [30 40] [50 60]])
```

e_f:
```
(Arr (-> ((Arr (Shp) Int)
          (Arr (Shp) Int))
         (Arr (Shp) Int))
     (Shp))
```

$$\frac{\Theta;\Delta;\Gamma \vdash e_f : (\texttt{Arr}\ \iota_f\ (\texttt{->}\ ((\texttt{Arr}\ \iota_i\ \tau_i)\ \cdots)\ (\texttt{Arr}\ \iota_o\ \tau_o))) \quad \Theta;\Delta;\Gamma \vdash e_{aj} : (\texttt{Arr}\ (\texttt{++}\ \iota_{aj}\ \iota_{ij})\ \tau_i)\ \text{ for each } j \quad \iota_p = \bigsqcup [\![\iota_f\ \iota_a\ \cdots]\!]}{\Theta;\Delta;\Gamma \vdash (e_f\ e_a\ \cdots) : (\texttt{Arr}\ (\texttt{++}\ \iota_p\ \iota_o)\ \tau_o)}\ \text{T-App}$$

```
(+ [1 2 3] [[10 20] [30 40] [50 60]])
```

e_f:
```
(Arr (-> ((Arr (Shp) Int)
          (Arr (Shp) Int))
         (Arr (Shp) Int))
     (Shp))
```

e_{a1}:
```
(Arr (Shp 3) Int)
```

e_{a2}:
```
(Arr (Shp 3 2) Int)
```

$$\frac{\Theta;\Delta;\Gamma \vdash e_f : (\texttt{Arr}\ \iota_f\ (\texttt{->}\ ((\texttt{Arr}\ \iota_i\ \tau_i)\ \cdots)\ (\texttt{Arr}\ \iota_o\ \tau_o))) \quad \Theta;\Delta;\Gamma \vdash e_{aj} : (\texttt{Arr}\ (\texttt{++}\ \iota_{aj}\ \iota_{ij})\ \tau_i)\ \text{ for each } j \quad \iota_p = \bigsqcup [\![\iota_f\ \iota_a\ \cdots]\!]}{\Theta;\Delta;\Gamma \vdash (e_f\ e_a\ \cdots) : (\texttt{Arr}\ (\texttt{++}\ \iota_p\ \iota_o)\ \tau_o)}\ \text{T-App}$$

```
(+ [1 2 3] [[10 20] [30 40] [50 60]])
```

e_f:
```
(Arr (-> ((Arr (Shp) Int)
          (Arr (Shp) Int))
         (Arr (Shp) Int))
     (Shp))
```

e_{a1}:
```
(Arr (Shp 3) Int)
```

e_{a2}:
```
(Arr (Shp 3 2) Int)
```

$$\iota_p = \texttt{(Shp 3 2)}$$

$$\frac{\Theta;\Delta;\Gamma \vdash e_f : (\texttt{Arr}\ \iota_f\ (\texttt{->}\ ((\texttt{Arr}\ \iota_i\ \tau_i)\ \cdots)\ (\texttt{Arr}\ \iota_o\ \tau_o))) \quad \Theta;\Delta;\Gamma \vdash e_{aj} : (\texttt{Arr}\ (\texttt{++}\ \iota_{aj}\ \iota_{ij})\ \tau_i)\ \text{ for each } j \quad \iota_p = \bigsqcup [\![\iota_f\ \iota_a\ \cdots]\!]}{\Theta;\Delta;\Gamma \vdash (e_f\ e_a\ \cdots) : (\texttt{Arr}\ (\texttt{++}\ \iota_p\ \iota_o)\ \tau_o)}\ \text{T-App}$$

```
(+ [1 2 3] [[10 20] [30 40] [50 60]])
      : (Arr (Shp 3 2) Int)
```

e_f: `(Arr (-> ((Arr (Shp) Int) (Arr (Shp) Int)) (Arr (Shp) Int)) (Shp))`

e_{a1}: `(Arr (Shp 3) Int)`

e_{a2}: `(Arr (Shp 3 2) Int)`

$$\iota_p = \texttt{(Shp 3 2)}$$

$$\frac{\begin{array}{c}\Theta; \Delta; \Gamma \vdash e_s : \texttt{(Arr (Shp) (Sigma (}(x_i' \ \gamma) \ \cdots \texttt{)} \ \tau_s\texttt{))} \\ \Theta, x_i :: \gamma \ \cdots ; \Delta; \Gamma, x_e : \tau_s[x_i' \mapsto x_i, \cdots] \vdash e_b : \tau_b \\ \Theta; \Delta \vdash \tau_b :: \texttt{Array}\end{array}}{\Theta; \Delta; \Gamma \vdash \texttt{(unbox (}x_i \ \cdots \ \ x_e \ e_s\texttt{)} \ e_b\texttt{)} : \tau_b} \quad \textsc{T-Unbox}$$

$$\frac{\begin{array}{c}\Theta; \Delta; \Gamma \vdash e_s : \texttt{(Arr (Shp) (Sigma (}(x_i' \ \gamma) \ \cdots\texttt{)}\ \tau_s\texttt{))} \\ \Theta, x_i :: \gamma \ \cdots\ ; \Delta; \Gamma, x_e : \tau_s[x_i' \mapsto x_i, \cdots] \vdash e_b : \tau_b \\ \Theta; \Delta \vdash \tau_b :: \texttt{Array}\end{array}}{\Theta; \Delta; \Gamma \vdash \texttt{(unbox (}x_i \ \cdots\ \ x_e \ e_s\texttt{)}\ e_b\texttt{)} : \tau_b} \quad \text{T-UNBOX}$$

Ensure ragged result data typed appropriately

$$\frac{\Theta; \Delta; \Gamma \vdash t : \tau' \qquad \tau \cong \tau'}{\Theta; \Delta; \Gamma \vdash t : \tau} \quad \text{T-Eqv}$$

$$\frac{\Theta; \Delta; \Gamma \vdash t : \tau' \qquad \tau \cong \tau'}{\Theta; \Delta; \Gamma \vdash t : \tau} \quad \textsc{T-Eqv}$$

$$\frac{\tau \cong \tau' \qquad \textsc{Valid}\, [\![\iota = \iota']\!]}{(\texttt{Arr}\ \iota\ \tau) \cong (\texttt{Arr}\ \iota'\ \tau')} \quad \textsc{TEqv-Array}$$

$$\frac{\Theta; \Delta; \Gamma \vdash t : \tau' \qquad \tau \cong \tau'}{\Theta; \Delta; \Gamma \vdash t : \tau} \quad \text{T-Eqv}$$

$$\frac{\tau \cong \tau' \qquad \textsc{Valid}\ [\![\iota = \iota']\!]}{(\texttt{Arr}\ \iota\ \tau) \cong (\texttt{Arr}\ \iota'\ \tau')} \quad \text{TEqv-Array}$$

Canonicalize: flat append of bags-of-summands

Cell-type polymorphism

```
vec-norm :

  (-> ((Arr [3] Float))
      Float)
```

```
vec-norm :

  (-> ((Arr [2] Float))
      Float)
```

```
vec-norm :

  (-> ((Arr [L] Float))
      Float)
```

```
vec-norm :

(Pi ((L Dim))
  (-> ((Arr [L] Float))
      Float))
```

```
vec-norm :

(Pi ((L Dim))
  (-> ((Arr [L] Float))
      Float))
```

n.b., shorthand for

```
(Arr [] (Pi ((L Dim))
          (Arr [] (-> ((Arr [L] Float))
                      (Arr [] Float)))))
```

m^*

```
m*      (Pi ((L Dim) (M Dim) (N Dim))
          (-> ((Arr [L M] Float)
               (Arr [M N] Float))
              (Arr [L N] Float)))
```

```
m*      (Pi ((L Dim) (M Dim) (N Dim))
          (-> ((Arr [L M] Float)
               (Arr [M N] Float))
              (Arr [L N] Float)))

head
```

```
m*      (Pi ((L Dim) (M Dim) (N Dim))
          (-> ((Arr [L M] Float)
               (Arr [M N] Float))
              (Arr [L N] Float)))

head    (Pi ((L Dim))
          (∀ ((T Atom))
            (-> ((Arr [(+ L 1)] T))
                (Arr [] T))))
```

```
m*      (Pi ((L Dim) (M Dim) (N Dim))
          (-> ((Arr [L M] Float)
               (Arr [M N] Float))
              (Arr [L N] Float)))

head    (Pi ((L Dim))
          (∀ ((T Atom))
            (-> ((Arr [(+ L 1)] T))
                (Arr [] T))))

append
```

```
m*      (Pi ((L Dim) (M Dim) (N Dim))
          (-> ((Arr [L M] Float)
               (Arr [M N] Float))
              (Arr [L N] Float)))

head    (Pi ((L Dim))
          (∀ ((T Atom))
            (-> ((Arr [(+ L 1)] T))
                (Arr [] T))))

append  (Pi ((L1 Dim) (L2 Dim) (C Shape))
          (∀ ((T Atom))
            (-> ((Array (++ [L1] C) T)
                 (Array (++ [L2] C) T))
                (Array (++ [(+ L1 L2)] C) T))))
```

Typing ragged data

```
(box [1 2 3 4])
```

```
(box [1 2 3 4])
```

```
(box 4 [1 2 3 4]
     (Sigma ((N Dim))
       (Arr [N] Int)))
```

```
(box [1 2 3 4])
```

```
(box 4 [1 2 3 4]
     (Sigma ((N Dim))
       (Arr [N] Int)))

(box 3 [1 2 3 4]
     (Sigma ((N Dim))
       (Arr [(+ 1 N)] Int)))
```

```
(box [1 2 3 4])
```

```
(box 4 [1 2 3 4]
     (Sigma ((N Dim))
       (Arr [N] Int)))

(box 3 [1 2 3 4]
     (Sigma ((N Dim))
       (Arr [(+ 1 N)] Int)))

(box [4] [1 2 3 4]
     (Sigma ((S Shape))
       (Arr S Int)))
```

```
(Sigma ((N Dim))
  (Arr [N] Int))
```

```
(Sigma ((N Dim))
  (Arr [N] Int))
```

Vector, unknown length

```
(Sigma ((N Dim))
  (Arr [N] Int))

(Sigma ((N Dim))
  (Arr [(+ 1 N)] Int))
```

Vector, unknown length

```
(Sigma ((N Dim))
  (Arr [N] Int))
```

Vector, unknown length

```
(Sigma ((N Dim))
  (Arr [(+ 1 N)] Int))
```

Vector, nonempty

```
(Sigma ((N Dim))
  (Arr [N] Int))
```

Vector, unknown length

```
(Sigma ((N Dim))
  (Arr [(+ 1 N)] Int))
```

Vector, nonempty

```
(Sigma ((M Dim) (N Dim))
  (Arr [M N] Int))
```

```
(Sigma ((N Dim))
  (Arr [N] Int))
```

Vector, unknown length

```
(Sigma ((N Dim))
  (Arr [(+ 1 N)] Int))
```

Vector, nonempty

```
(Sigma ((M Dim) (N Dim))
  (Arr [M N] Int))
```

Matrix, unknown dimensions

```
(Sigma ((N Dim))
  (Arr [N] Int))
```

Vector, unknown length

```
(Sigma ((N Dim))
  (Arr [(+ 1 N)] Int))
```

Vector, nonempty

```
(Sigma ((M Dim) (N Dim))
  (Arr [M N] Int))
```

Matrix, unknown dimensions

```
(Sigma ((S Shape))
  (Arr S Int))
```

```
(Sigma ((N Dim))
  (Arr [N] Int))
```

Vector, unknown length

```
(Sigma ((N Dim))
  (Arr [(+ 1 N)] Int))
```

Vector, nonempty

```
(Sigma ((M Dim) (N Dim))
  (Arr [M N] Int))
```

Matrix, unknown dimensions

```
(Sigma ((S Shape))
  (Arr S Int))
```

Completely unknown shape

```
(Sigma ((N Dim))
  (Arr [N] Int))
```

Vector, unknown length

```
(Sigma ((N Dim))
  (Arr [(+ 1 N)] Int))
```

Vector, nonempty

```
(Sigma ((M Dim) (N Dim))
  (Arr [M N] Int))
```

Matrix, unknown dimensions

```
(Sigma ((S Shape))
  (Arr S Int))
```

Completely unknown shape

```
(Sigma ((N Dim) (S Shape))
  (Arr (++ [N] S) Int))
```

```
(Sigma ((N Dim))
  (Arr [N] Int))
```

Vector, unknown length

```
(Sigma ((N Dim))
  (Arr [(+ 1 N)] Int))
```

Vector, nonempty

```
(Sigma ((M Dim) (N Dim))
  (Arr [M N] Int))
```

Matrix, unknown dimensions

```
(Sigma ((S Shape))
  (Arr S Int))
```

Completely unknown shape

```
(Sigma ((N Dim) (S Shape))
  (Arr (++ [N] S) Int))
```

Rank at least one

```
(Sigma ((N Dim))
  (Arr [N] Int))
```
Vector, unknown length

```
(Sigma ((N Dim))
  (Arr [(+ 1 N)] Int))
```
Vector, nonempty

```
(Sigma ((M Dim) (N Dim))
  (Arr [M N] Int))
```
Matrix, unknown dimensions

```
(Sigma ((S Shape))
  (Arr S Int))
```
Completely unknown shape

```
(Sigma ((N Dim) (S Shape))
  (Arr (++ [N] S) Int))
```
Rank at least one

But still all regular

```
(Arr [4]
  (Sigma ((N Dim))
    (Arr [N] Int)))
```

```
(Arr [4]
  (Sigma ((N Dim))
    (Arr [N] Int)))
```

Four vectors, may vary in length

```
(Arr [4]
  (Sigma ((N Dim))
    (Arr [N] Int)))

(Sigma ((N Dim))
  (Arr [4 N] Int))
```

Four vectors, may vary in length

```
(Arr [4]
  (Sigma ((N Dim))
    (Arr [N] Int)))
```

Four vectors, may vary in length

```
(Sigma ((N Dim))
  (Arr [4 N] Int))
```

Four vectors, same unknown length

```
(Arr [4]
  (Sigma ((N Dim))
    (Arr [N] Int)))
```

Four vectors, may vary in length

```
(Sigma ((N Dim))
  (Arr [4 N] Int))
```

Four vectors, same unknown length

```
(Arr [4]
  (Sigma ((S Shape))
    (Arr S Int)))
```

```
(Arr [4]
  (Sigma ((N Dim))
    (Arr [N] Int)))
```

Four vectors, may vary in length

```
(Sigma ((N Dim))
  (Arr [4 N] Int))
```

Four vectors, same unknown length

```
(Arr [4]
  (Sigma ((S Shape))
    (Arr S Int)))
```

Four arrays which
may even differ in rank

```
(Arr [4]
  (Sigma ((N Dim))
    (Arr [N] Int)))
```

Four vectors, may vary in length

```
(Sigma ((N Dim))
  (Arr [4 N] Int))
```

Four vectors, same unknown length

```
(Arr [4]
  (Sigma ((S Shape))
    (Arr S Int)))
```

Four arrays which
may even differ in rank

```
(Sigma ((S Shape))
  (Arr (++ [4] S) Int))
```

```
(Arr [4]
  (Sigma ((N Dim))
    (Arr [N] Int)))
```

Four vectors, may vary in length

```
(Sigma ((N Dim))
  (Arr [4 N] Int))
```

Four vectors, same unknown length

```
(Arr [4]
  (Sigma ((S Shape))
    (Arr S Int)))
```

Four arrays which
may even differ in rank

```
(Sigma ((S Shape))
  (Arr (++ [4] S) Int))
```

Four arrays with
same unknown shape

```
(Arr [4]
  (Sigma ((N Dim))
    (Arr [N] Int)))
```

Four vectors, may vary in length

```
(Sigma ((N Dim))
  (Arr [4 N] Int))
```

Four vectors, same unknown length

```
(Arr [4]
  (Sigma ((S Shape))
    (Arr S Int)))
```

Four arrays which
may even differ in rank

```
(Sigma ((S Shape))
  (Arr (++ [4] S) Int))
```

Four arrays with
same unknown shape

Raggedness is frame structure around shape uncertainty

```
filter : (Pi ((L Dim) (C Shape))
           (∀ ((T Atom))
             (-> ((Arr [L] Bool)
                  (Arr (++ [L] C) T))
                 (Sigma ((N Dim))
                   (++ [N] C) T))))
```

```
filter : (Pi ((L Dim) (C Shape))
           (∀ ((T Atom))
             (-> ((Arr [L] Bool)
                  (Arr (++ [L] C) T))
                 (Sigma ((N Dim))
                   (++ [N] C) T))))

filter/p : (Pi ((L Dim) (C Shape))
           (∀ ((T Atom))
             (-> ((-> ((Arr C T)) Bool)
                  (Arr (++ [L] C) T))
                 (Sigma ((N Dim))
                   (++ [N] C) T))))
```

```
; Build a predicate to check
; membership in [lo,hi)
in-range? : (-> (Int Int)
                (-> Int Bool))
```

```
; Build a predicate to check
; membership in [lo,hi)
in-range? : (-> (Int Int)
                (-> Int Bool))


(define binned-grades
  (filter/p
   (in-range? [ 0 60 70 80  90]
              [60 70 80 90 101])
   grades))
```

```
; Build a predicate to check
; membership in [lo,hi)
in-range? : (-> (Int Int)
                (-> Int Bool))


(define binned-grades
  (filter/p
   (in-range? [ 0 60 70 80  90]
              [60 70 80 90 101])
   grades))

binned-grades :
(Arr [5] (Sigma ((N Dim)) (Arr [N] Int)))
```

Type inference:

Identify index arguments elided by programmer

Type inference:

Identify index arguments elided by programmer

Function type

```
(Pi (e ...)
  (-> ((Arr ι σ)
       ...)
      τ))
```

Type inference:

Identify index arguments elided by programmer

Function type

```
(Pi (e ...)
  (-> ((Arr ι σ)
       ...)
      τ))
```

Argument types

```
(Arr κ σ) ...
```

Type inference:

Identify index arguments elided by programmer

Function type

```
(Pi (e ...)
  (-> ((Arr ι σ)
       ...)
      τ))
```

Argument types

```
(Arr κ σ) ...
```

- ∀ quantifiers for bound index vars

Type inference:

Identify index arguments elided by programmer

Function type

```
(Pi (e ...)
  (-> ((Arr ι σ)
       ...)
      τ))
```

Argument types

```
(Arr κ σ) ...
```

- ∀ quantifiers for bound index vars
- ∃ quantifiers for frames

Type inference:

Identify index arguments elided by programmer

Function type

```
(Pi (e ...)
  (-> ((Arr ι σ)
       ...)
      τ))
```

Argument types

```
(Arr κ σ) ...
```

- ∀ quantifiers for bound index vars
- ∃ quantifiers for frames
- ∃ quantifiers for index arguments

Type inference:

Identify index arguments elided by programmer

Function type

```
(Pi (e ...)
  (-> ((Arr ι σ)
       ...)
      τ))
```

Argument types

```
(Arr κ σ) ...
```

- ∀ quantifiers for bound index vars
- ∃ quantifiers for frames (eliminated before)
- ∃ quantifiers for index arguments

Type inference:

Identify index arguments elided by programmer

Function type

```
(Pi (e ...)
  (-> ((Arr ι σ)
       ...)
      τ))
```

Argument types

```
(Arr κ σ) ...
```

- ∀ quantifiers for bound index vars
- ∃ quantifiers for frames (eliminated before)
- ∃ quantifiers for index arguments (solving for these)

Function type

```
(Pi (e ...)
  (-> ((Arr ι σ)
       ...)
      τ))
```

Argument types

```
(Arr κ σ) ...
```

Function type

```
(Pi (e ...)
  (-> ((Arr ι σ)
       ...)
      τ))
```

Argument types

```
(Arr κ σ) ...
```

$$\forall \overrightarrow{x} . \exists \overrightarrow{f}, \overrightarrow{e} . \bigwedge_{i=1}^{n} (f_i +\!\!+ \iota_i \equiv \kappa_i)$$

Function type

```
(Pi (e ...)
  (-> ((Arr ι σ)
       ...)
      τ))
```

Argument types

```
(Arr κ σ) ...
```

$$\forall \overrightarrow{x} . \exists \overrightarrow{f}, \overrightarrow{e} . \bigwedge_{i=1}^{n} (f_i +\!\!+ \iota_i \equiv \kappa_i)$$

$$\mathcal{I} \quad = \quad e_i \mapsto T_i(\overrightarrow{x})$$

Function type

```
(Pi (e ...)
  (-> ((Arr ι σ)
       ...)
      τ))
```

Argument types

```
(Arr κ σ) ...
```

$$\forall \overrightarrow{x} . \exists \overrightarrow{f}, \overrightarrow{e} . \bigwedge_{i=1}^{n} (f_i +\!\!+ \iota_i \equiv \kappa_i)$$

$$\mathcal{I} \quad = \quad e_i \mapsto T_i(\overrightarrow{x}) \qquad\qquad \Sigma = +\!\!+, \square, +$$

Function type

```
(Pi (e ...)
  (-> ((Arr ι σ)
       ...)
      τ))
```

Argument types

```
(Arr κ σ) ...
```

$$\forall \overrightarrow{x} . \exists \overrightarrow{f}, \overrightarrow{e} . \bigwedge_{i=1}^{n} (f_i +\!\!+ \iota_i \equiv \kappa_i)$$

$$\mathcal{I} \quad = \quad e_i \mapsto T_i(\overrightarrow{x}) \qquad\qquad \Sigma = +\!\!+, \square, +$$

Pretend $\overrightarrow{x}$ are additional generators

Why can we treat $\overrightarrow{x}$ like additional generators?

Why can we treat $\overrightarrow{x}$ like additional generators?

Algorithm for existential fragment (Makanin, 1977)

$$c +\!\!+ [5] +\!\!+ d +\!\!+ [5] +\!\!+ d \equiv d +\!\!+ [5] +\!\!+ [2] +\!\!+ d +\!\!+ e$$

Why can we treat $\overrightarrow{x}$ like additional generators?

Algorithm for existential fragment (Makanin, 1977)

$$c +\!\!+ [5] +\!\!+ d +\!\!+ [5] +\!\!+ d \equiv d +\!\!+ [5] +\!\!+ [2] +\!\!+ d +\!\!+ e$$

"How might subsequences align with each other?"

Why can we treat $\overrightarrow{x}$ like additional generators?

Algorithm for existential fragment (Makanin, 1977)

$$c +\!\!+ [5] +\!\!+ d +\!\!+ [5] +\!\!+ d \equiv d +\!\!+ [5] +\!\!+ [2] +\!\!+ d +\!\!+ e$$

"How might subsequences align with each other?"

Equations about smaller sequences

Why can we treat $\overrightarrow{x}$ like additional generators?

Algorithm for existential fragment (Makanin, 1977)

$$c +\!\!+ [5] +\!\!+ d +\!\!+ [5] +\!\!+ d \equiv d +\!\!+ [5] +\!\!+ [2] +\!\!+ d +\!\!+ e$$

"How might subsequences align with each other?"

Equations about smaller sequences

No boundaries *inside* a generator

Why can we treat $\overrightarrow{x}$ like additional generators?

Algorithm for existential fragment (Makanin, 1977)

$$c +\!\!+ [5] +\!\!+ d +\!\!+ [5] +\!\!+ d \equiv d +\!\!+ [5] +\!\!+ [2] +\!\!+ d +\!\!+ e$$

"How might subsequences align with each other?"

Equations about smaller sequences

No boundaries *inside* a generator ... or a universal variable

Why can we treat $\overrightarrow{x}$ like additional generators?

Algorithm for existential fragment (Makanin, 1977)

$$c +\!\!+ [5] +\!\!+ d +\!\!+ [5] +\!\!+ d \equiv d +\!\!+ [5] +\!\!+ [2] +\!\!+ d +\!\!+ e$$

"How might subsequences align with each other?"

Equations about smaller sequences

No boundaries *inside* a generator ... or a universal variable

Caveat: only works in conjunctive fragment

Type inference

Type inference

Compilation

Type inference

Compilation

Implement sketched algorithm

Type inference

Implement sketched algorithm

Codebase for evaluating inference

Compilation

Type inference

Implement sketched algorithm

Codebase for evaluating inference

Compilation

Possible targets?

Type inference

Implement sketched algorithm

Codebase for evaluating inference

Compilation

Possible targets?

IRs for array manipulation

Type inference

Implement sketched algorithm

Codebase for evaluating inference

Compilation

Possible targets?

IRs for array manipulation

Broad goal

How much can we accomplish by combining a few powerful features instead of building lots of purpose-specific machinery?