Companion to Bird & de Moor, Algebra of Programming

§0. Laws from Relation Algebra

§0.1. Adjunctions

(0.1a)

	F⊣G	F's type	monad FG	𝟙⊑FG	GF⊑𝟙	F preserves ∪	G preserves ∩	FGF=F	GFG=G
1	◁⊣▷	A→ A⊗A	◁▷=𝟙	𝟙⊑◁▷	▷◁⊑𝟙	(R ∪ S)◁= R◁ ∪ S◁	(R∩S)▷= R▷∩S▷	◁▷◁=◁	▷◁▷=▷
2	⊸⊣⊸	A→𝕀	⊸⊸=T	𝟙⊑⊸⊸	⊸⊸⊑𝟙	(R ∪ S)⊸= R⊸ ∪ S⊸	(R∩S)⊸= R⊸∩S⊸	⊸⊸⊸=⊸	⊸⊸⊸=⊸
3	°⊣°	(A→B)→ (B→A)	R°° = R	R°° = R	R°° = R	(R ∪ S)° = S° ∪ R°	(R ∩ S)° = R° ∩ S°	R°° = R	R°° = R
4	⊸◁⊣▷⊸	(X⊗A→Y)→ (X→Y⊗A)	𝟙	(⊸◁⊗𝟙) (𝟙⊗▷⊸)=𝟙	(𝟙⊗⊸◁) (▷⊸⊗𝟙)=𝟙	=	=	=	=
5	▷⊸⊣⊸◁	(X→Y⊗A)→ (X⊗A→Y)	𝟙	(𝟙⊗⊸◁) (▷⊸⊗𝟙)=𝟙	(⊸◁⊗𝟙) (𝟙⊗▷⊸)=𝟙	=	=	=	=
6	Δ⊣∩	(A→B)→ (A→B)²	R ∩ R = R	R ∩ R = R	R ∩ S ⊑ R	Δ(R ∪ S)= ΔR ∪ ΔS	(R∩T)∩(S∩U)= (R∩S)∩(T∩U)	R ∩ R = R	R ∩ R = R
7	∪⊣Δ	(A→B)²→ (A→B)	(R,S)↦ (R ∪ S,R ∪ S)	R ⊑ R ∪ S	R ∪ R = R	(R ∪ T) ∪ (S ∪ U)= (R ∪ S) ∪ (T ∪ U)	Δ(R∩S)= ΔR∩ΔS	R ∪ R = R	R ∪ R = R
8	𝟘⊣!	{*}→ (A→B)			𝟘 ⊑ R				
9	·S⊣/S	(A→B)→ (A→C)	S/S	R⊑(RS)/S	(T/S)S⊑T	(S ∪ T) R = (S R) ∪ (T R)	(R ∩ R') / S = (R / S) ∩ (R' / S)	((RS)/S)S =RS	((T/S)S)/S =T/S
10	S·⊣S\	(B→C)→ (A→C)	S\S	R⊑S\(SR)	S(S\T)⊑T	R (S ∪ T) = (R S) ∪ (R T)	S \ (R ∩ R') = (S \ R) ∩ (S \ R')	S(S\(SR)) =SR	S\(S(S\T)) =S\T
11	R∩⊣R⇒	(A→B)→ (A→B)	X↦R⇒(X∩R)	X⊑R⇒(X∩R)	R∩(R⇒Y)⊑Y	R∩(X ∪ Y)= (R∩X) ∪ (R∩Y)	R ⇨ S ∩ T = (R ⇨ S) ∩ (R ⇨ T)	R∩(R⇒(X∩R)) =X∩R	R⇒(R∩(R⇒Y)) =R⇒Y
12	𝒟⊣·T	(A→B)→ Cor A	R↦(𝒟R)T	R⊑(𝒟R)T	𝒟(AT)⊑A	𝒟(R ∪ S)= 𝒟R ∪ 𝒟S	(A∩B)T= AT∩BT	𝒟((𝒟R)T) =𝒟R	(𝒟(AT))T =AT
13	ℛ⊣T·	(A→B)→ Cor B	R↦T(ℛR)	R⊑T(ℛR)	ℛ(TA)⊑A	ℛ(R ∪ S)= ℛR ∪ ℛS	T(A∩B)= TA∩TB	ℛ(T(ℛR)) =ℛR	T(ℛ(TA)) =TA
14	·f⊣·f°	(A→B)→ (A→C)	ff°	Map(f) ⟹ 𝟙 ⊑ f f°	f°f⊑𝟙	(S ∪ T) R = (S R) ∪ (T R)	Simple(g) ⟹ (X ∩ Y) g° = (X g°) ∩ (Y g°)	ff°f=f	f°ff°=f°
15	f°·⊣f·	(A→C)→ (B→C)	ff°	Map(f) ⟹ 𝟙 ⊑ f f°	f°f⊑𝟙	R (S ∪ T) = (R S) ∪ (R T)	f(X∩Y)= fX∩fY	ff°f=f	f°ff°=f°
16	i⊣E	Map↪Rel	E	$\frac{\mathbb{1}}{\ni}$:A→PA	∋:PB→B	—	—	$\frac{\ni}{\ni}$=𝟙	$\frac{R}{\ni}$∋=R
17	·∋⊣ $\frac{}{\ni}$	Map(A,PB)→ (A→B)	𝟙	$\frac{f\ni}{\ni}$=f	$\frac{R}{\ni}$∋=R	—	—	$\frac{\frac{f\ni}{\ni}}{\ni}$ =f∋	$\frac{\frac{R}{\ni}\ni}{\ni}$ =$\frac{R}{\ni}$
18	$\frac{}{\ni}$ ⊣·∋	(A→B)→ Map(A,PB)	𝟙	$\frac{R}{\ni}$∋=R	$\frac{f\ni}{\ni}$=f	—	—	$\frac{\frac{R}{\ni}\ni}{\ni}$ =$\frac{R}{\ni}$	$\frac{f\ni}{\ni}$∋ =f∋

§0.2. Composing adjunctions

(0.2a)

		·T	T·	·g	g°·	Tn	°	⊶◁	Δ
1	·S	R / (S₁ S₂) = R / S₂ / S₁	S \ (R / W) = (S \ R) / W	R/(Sg)= (Rg°)/S	Map(f) ⟹ f (R / S) = (f R) / S	—	(Y/S)°= S°\(Y°)	(RS)^= R^(S⊗𝟙)	(R ∩ R') / S = (R / S) ∩ (R' / S)
2	S·	S \ (R / W) = (S \ R) / W	(S₁ S₂) \ R = S₂ \ (S₁ \ R)	S\(Rg°)= (S\R)g°	(g°S)\R= S\(gR)	—	(S\Y)°= Y°/S°	((S⊗𝟙)R)^= SR^	S \ (R ∩ R') = (S \ R) ∩ (S \ R')
3	·f	Map(f) ⟹ R / (f S) = (R / S) f°	T\(Rf°)= (T\R)f°	(R S)° = S° R°			(R S)° = S° R°		Simple(g) ⟹ (X ∩ Y) g° = (X g°) ∩ (Y g°)
4	f°·	Map(f) ⟹ f (R / S) = (f R) / S	(Tf°)\R= f(T\R)	—	(R S)° = S° R°		(R S)° = S° R°		f(T₁∩T₂)= fT₁∩fT₂
5	Rn	—	—	—	—	R ⇨ S ⇨ T = R ∩ S ⇨ T	(R⇨Y)°= R°⇨(Y°)	—	R ⇨ S ∩ T = (R ⇨ S) ∩ (R ⇨ T)
6	°	(Y/T)°= T°\(Y°)	(T\Y)°= Y°/T°	(R S)° = S° R°	(R S)° = S° R°	(T⇨Y)°= T°⇨(Y°)	R°° = R	(R^)°= (R°⊗𝟙) (𝟙⊗▷⊶) both bends: the ° section's display	(T₁∩T₂)°= T₁°∩T₂°
7	⊶◁	(RT)^= R^(T⊗𝟙)	((T⊗𝟙)R)^= TR^	—	—	—	(R^)°= (R°⊗𝟙) (𝟙⊗▷⊶) both bends: the ° section's display	—	—
8	∪	(S ∪ T) R = (S R) ∪ (T R)	R (S ∪ T) = (R S) ∪ (R T)	(S ∪ T) R = (S R) ∪ (T R)	R (S ∪ T) = (R S) ∪ (R T)	R ∩ (S ∪ T) = R ∩ S ∪ R ∩ T	(R ∪ S)° = S° ∪ R°		
9	𝟘	𝟘 R = 𝟘	R 𝟘 = 𝟘	𝟘 R = 𝟘	R 𝟘 = 𝟘	R ∩ 𝟘 = 𝟘	𝟘° = 𝟘		

§0.3. Adjoint triples

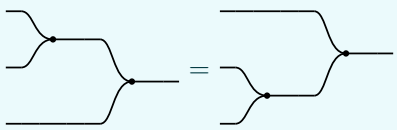
(0.3a)

```
·f⊣·f°⊣/f°
f°·⊣f·⊣f\
```

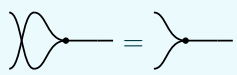
§0.4. Frobenius

Frobenius.

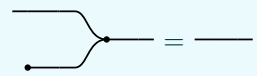
(0.4a)



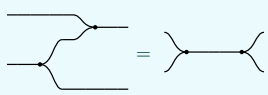
▷ associative



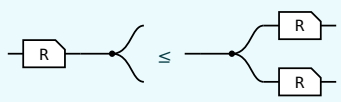
▷ commutative



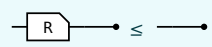
⊶ is its unit



Frobenius, one half — the other is its °

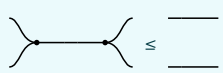


R ◁ ≤ ◁ (R ⊗ R)

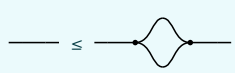


R ⊸ ≤ ⊸

(0.4b)



▷ ◁ ≤ 𝟙



𝟙 ≤ ◁ ▷

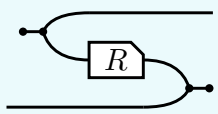
⊶ ⊸ ≤ 𝟙

𝟙 ≤ ⊸ ⊶

§0.5. ° is a contravariant 2-functor

(0.5a)

° is primitive, part of the data the first section lists. It turns both of R's wires round, and the Frobenius structure DRAWS that — the picture and the formula below are R°, not its definition:



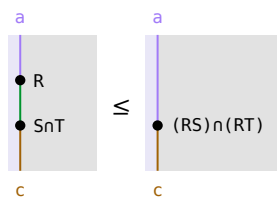
$$R^\circ = (\multimap\!\triangleleft \otimes \mathbb{1})(\mathbb{1} \otimes R \otimes \mathbb{1})(\mathbb{1} \otimes \triangleright\!\multimap)$$

where $\multimap\!\triangleleft : \mathbb{I} \to a \otimes a$ opens a pair of wires out of nothing and $\triangleright\!\multimap : b \otimes b \to \mathbb{I}$ closes one, so the input of R° is where the output of R was. Taking that formula as the definition would now be circular: $\triangleleft$ and $\multimap$ are $\triangleright^\circ$ and $\multimap^\circ$. The laws of ° are that it is a contravariant 2-functor $^\circ : \mathcal{C}^{op} \to \mathcal{C}$:

(i) $\mathbb{1}^\circ = \mathbb{1}$ (ii) $(RS)^\circ = S^\circ R^\circ$ (iii) $(R \otimes S)^\circ = R^\circ \otimes S^\circ$ (iv) $R \leq S$ implies $R^\circ \leq S^\circ$

§0.6. Meet and composition

(0.6a)

semi-distributivity, and what supplies it	picture
R (S∩T)⊆RS∩RT — the lax copy law. Equality exactly when R is single valued: the Maps section's F(R∩S)=FR∩FS.	

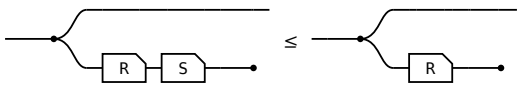
§0.7. Domain and range

(0.7a)

```
dom(R) ≜ 𝟙 ∩ (R R°)
Coreflexive(X) ⟹ dom(R) ⊑ X ⟺ R ⊑ X R
dom(R S) ⊑ dom(R)
dom(R ∩ S) = 𝟙 ∩ (S R°)
R entire Entire(R) ≜ dom(R) = 𝟙
R simple Simple(R) ≜ R° R ⊑ 𝟙
R a map Map(R) ≜ Entire(R) ∧ Simple(R)
Entire(R) ∧ Entire(S) ⟹ Entire(R S)
Simple(f) ∧ Simple(g) ⟹ Simple(f g)
Map(f) ∧ Map(g) ⟹ Map(f g)
Entire(R S) ⟹ Entire(R)
```

§0.8. Sliding the discard

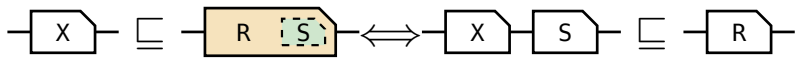
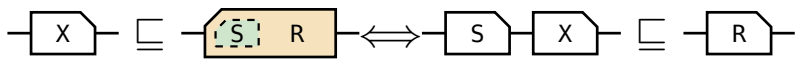
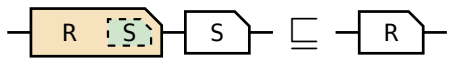
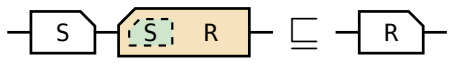
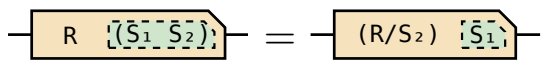
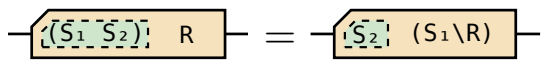
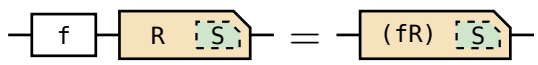
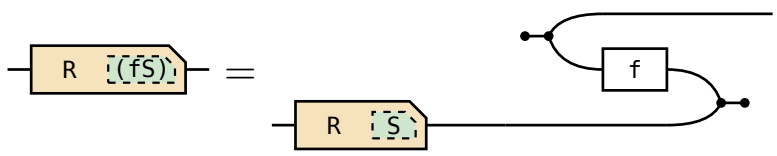
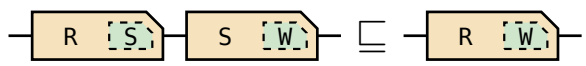
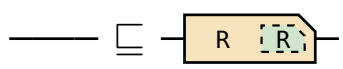
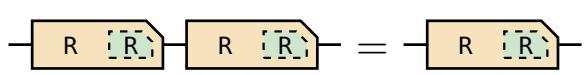
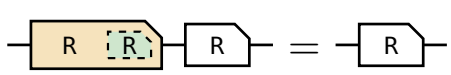
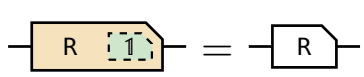
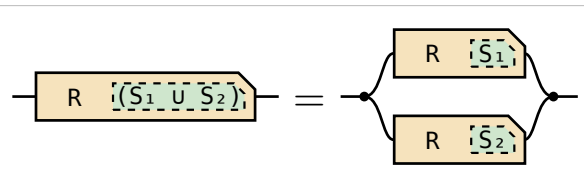
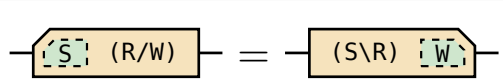
(0.8a)



S⊸⊑⊸, the lax axiom for ⊸ in the first section
— the discard slides back past S

§0.9. Division

(0.9a)

1	$X \sqsubseteq R/S \Longleftrightarrow XS \sqsubseteq R$ X is any x-to-y pairing; one that only pairs x with a y such that x admires everyone y hates lies inside R/S, and R/S is the largest such.	
2	$X \sqsubseteq S\backslash R \Longleftrightarrow SX \sqsubseteq R$ The mirror — divide on the left when x comes first.	
3	$(R/S)S \sqsubseteq R$ There is a y such that x admires everyone y hates, and p is one of the people y hates — then x admires p too. Strict at $S=\varnothing$: R/S is everyone, $(R/S)S=\varnothing$.	
4	$S\ (S\backslash R) \sqsubseteq R$ The mirror.	
5	**associate:** $R/(S_1S_2)=(R/S_2)/S_1$ **A friend's enemy** is two hops: divide by the far end first.	
6	$(S_1S_2)\backslash R=S_2\backslash(S_1\backslash R)$ The mirror.	
7	**maps:** $f\ (R/S)=(fR)/S$ Rename x before or after dividing — the licence to write fR/S.	
8	$R/(fS)=(R/S)f^\circ$ Rename y: a map leaves a denominator as f° outside the box.	
9	$(R/S)(S/W) \sqsubseteq R/W$ Someone who admires all of a hate-set that already covers everyone z works for admires those people too.	
10	$\mathbb{1} \sqsubseteq R/R$ R/R runs admirer to admirer: each admires everyone they admire. Strict: two people who each admire only a and b admire each other's idols too, and still stay two people.	
11	$(R/R)(R/R)=R/R$ R/R is the preorder **admires at least as much as**, and a preorder is idempotent. Freyd writes ⊑; with $\mathbb{1} \sqsubseteq R/R$ above it is an equality.	
12	$(R/R)R=R$ Reaching p through someone whose idols x fully admires is reaching p directly, since x admires their own idols.	
13	$R/\mathbb{1}=R$ Dividing by $\mathbb{1}$: p's set is just {p}, so admiring all of it is admiring p.	
14	$R/(S_1 \cup S_2)=R/S_1 \cap R/S_2$ Admiring a combined hate-set is admiring each set in full.	
15	$S\backslash(R/W)=(S\backslash R)/W$ Which is why $S\backslash R/W$ needs no bracket.	

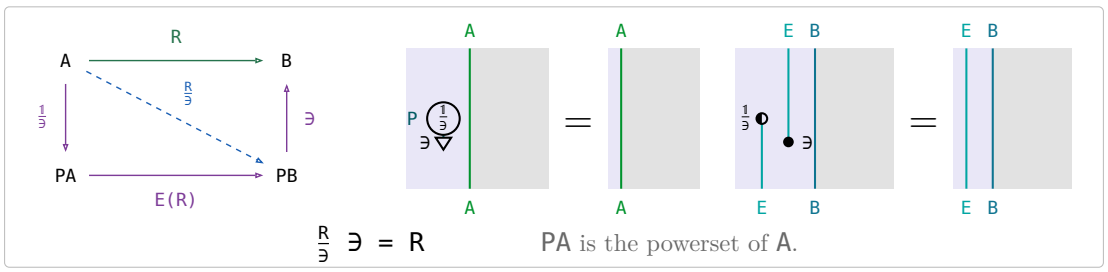
§0.10. Power allegories

(0.10a)

	law	the reading
1	$\ni_{\mathrm{R}}\square = R\square,\ \ni_{\mathrm{R}} = \ni_{\mathrm{R}\,\square}$	One ∋ per object, not per arrow.
2	∋ is **thick**	**Comprehension**: every x has a set of exactly the people x admires.
3	$\frac{R}{\ni}$: A→PB, for R : A→B	convert a relation to a function. a $\frac{R}{\ni}$ ={b|a R b}
4	Map($\frac{R}{\ni}$)	
5	$\frac{R}{\ni}$ ∋ = R	reading the chosen set back through ∋ returns the relation.
6	Simple(F) ⟹ F ⊑ $\frac{F\ \ni}{\ni}$	A partial choice of sets is inside the total one.
7	the **singleton map** is monic: $\frac{\mathbb{1}}{\ni}$ ($\frac{\mathbb{1}}{\ni}$)° ⊑ 𝟙 two points with the same one-person set are the same point.	The one-person set.
8	$\frac{\ni}{\ni}$ = 𝟙	Make the set of a set, then read it back one level down.
9	**fusion:** Map(f) ⟹ $\frac{f\ R}{\ni}$ = f $\frac{R}{\ni}$	Naturality of the unit, Map(f) ⟹ f $\frac{\mathbb{1}}{\ni}$ = $\frac{\mathbb{1}}{\ni}$ E(f) . renaming a point and then taking its one-person set is taking the set and renaming inside it.
10	Map(f) ∧ 𝟙□ = ∋□ ⟹ $\frac{f}{\ni}$ = f $\frac{\mathbb{1}}{\ni}$	Rename first or take singletons first — the fusion row above at R=𝟙.
11	R□ = ∋□ ⟹ R /ₛ S = $\frac{R}{\ni}$ ($\frac{S}{\ni}$)°	x and y match when R sends x and S sends y to the same set.
12	E(R) ≜ $\frac{\ni\ R}{\ni}$	E(R): PA→PB, image of a set of A
13	$\frac{R}{\ni}$ = $\frac{\mathbb{1}}{\ni}$ E(R)	$\frac{\mathbb{1}}{\ni}$: x↦{x}
14	$\frac{S}{\ni}$ E(R) = $\frac{S\ R}{\ni}$	absorption — the monad's composition law, $\frac{S}{\ni}$ ⋄ $\frac{R}{\ni}$ = $\frac{SR}{\ni}$, §$\mathcal{A}$≅Kleisli(E) in Relation Algebra
15	E(R) = E($\frac{R}{\ni}$) union	the image of a set under R is the union of the R-sets of its points.
16	⊇ ≜ ∋ / ∋	xs⊇ys⟺∀a. ys∋a→xs∋a
17	⊆ = ⊇°	xs⊆ys⟺ys⊇xs
18	T° $\frac{T}{\ni}$ ⊑ ∈	every T-value of a point is a member of its T-set: Λ(T) is a map and Λ(T)∋=T.

§0.11. i⊣E: power transpose through singleton

(0.11a)

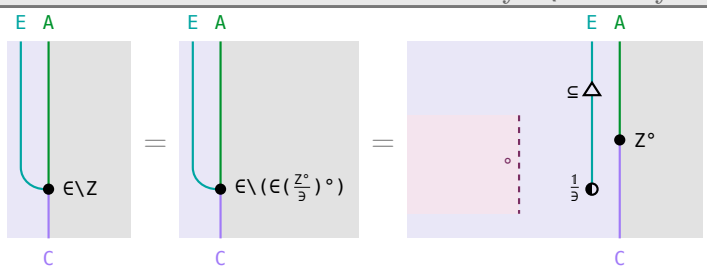


$\frac{R}{\ni} \ni = R$ PA is the powerset of A.

§0.12. ∈\ as a composite

$$\in \backslash Z = \subseteq (\tfrac{Z^\circ}{\ni})^\circ \tag{0.12a}$$

xs is related to **c** by ∈\Z exactly when **xs** ⊆ Λ(Z°)(c), the set of every **a** with **a** Z **c**



(1) (2) (3)

1	∈ \ Z	Z=∈Λ(Z°)° — R/∋ ∋ = R
2	= ∈ \ (∈ (Z°/∋)°)	X\(Yf°)=(X\Y)f°, f a map
3	= ⊆ (Z°/∋)°	

§1. Programs

§1.1. Datatypes

§1.2. Natural numbers

§1.3. Lists

§1.4. Trees

§1.5. Inverses

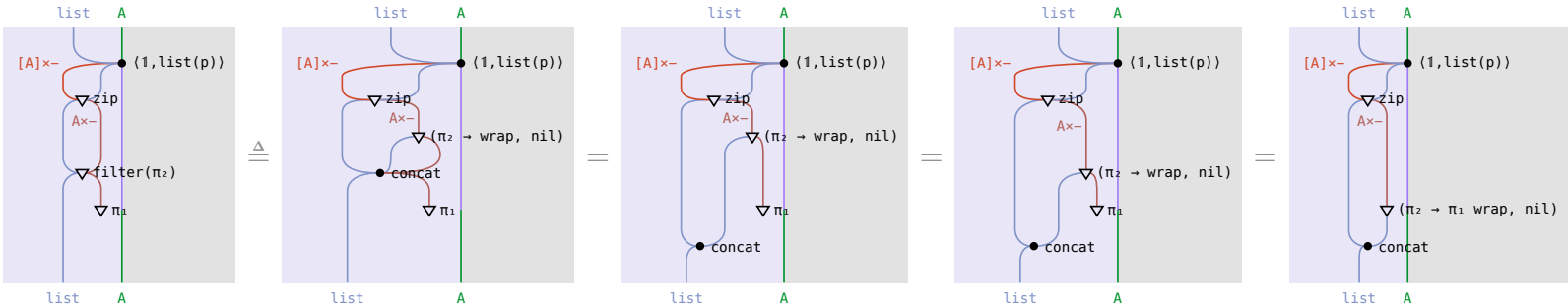
§1.6. Polymorphic functions

§1.7. Pointwise and point-free

⟨𝟙,list(p)⟩ zip filter(π₂) list(π₁) = filter(p) (1.7a)

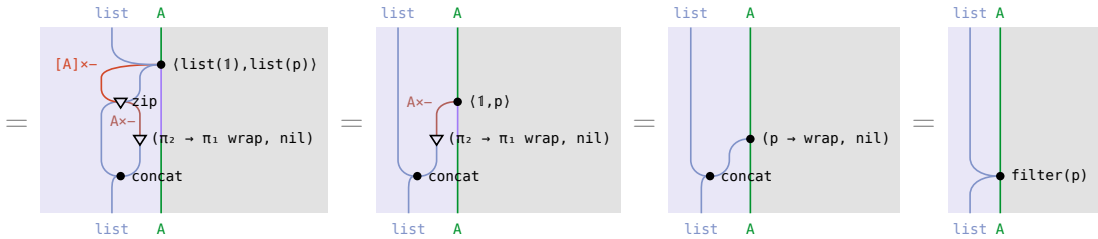
pairing each element with its answer under **p** (**⟨𝟙,list(p)⟩ zip**), keeping the pairs answered **true** (**filter(π₂)**) and dropping the answers (**list(π₁)**) is **filter(p)**

the starting composite



(1) (2) (3) (4) (5)

1	⟨𝟙,list(p)⟩ zip filter(π₂) list(π₁)	definition of filter: filter(p) ≜ list((p → wrap, nil)) concat
2	≜ ⟨𝟙,list(p)⟩ zip list((π₂ → wrap, nil)) concat list(π₁)	(1.6): concat list(f) = list(list(f)) concat
3	= ⟨𝟙,list(p)⟩ zip list((π₂ → wrap, nil)) list(list(π₁)) concat	(1.8) backwards: list(f) list(g) = list(fg)
4	= ⟨𝟙,list(p)⟩ zip list((π₂ → wrap, nil) list(π₁)) concat	(1.11): (p → f, g)h = (p → fh, gh); (1.5): wrap list(f) = f wrap; (1.4): nil list(f) = nil
5	= ⟨𝟙,list(p)⟩ zip list((π₂ → π₁ wrap, nil)) concat	(1.9) backwards: 𝟙 = list(𝟙)



(6) (7) (8) (9)

6	= ⟨list(𝟙),list(p)⟩ zip list((π₂ → π₁ wrap, nil)) concat	(1.7): ⟨list(f),list(g)⟩ zip = list(⟨f,g⟩) (1.8) backwards: list(f) list(g) = list(fg)
7	= list(⟨𝟙,p⟩ (π₂ → π₁ wrap, nil)) concat	(1.10): h(p → f, g) = (hp → hf, hg); (1.1): ⟨f,g⟩π₁ = f; (1.3): f nil = nil (1.12): 𝟙 f = f
8	= list((p → wrap, nil)) concat	definition of filter
9	= filter(p)	

§2. Functions and Categories

§2.1. Categories

§2.2. Functors

§2.3. Natural transformations

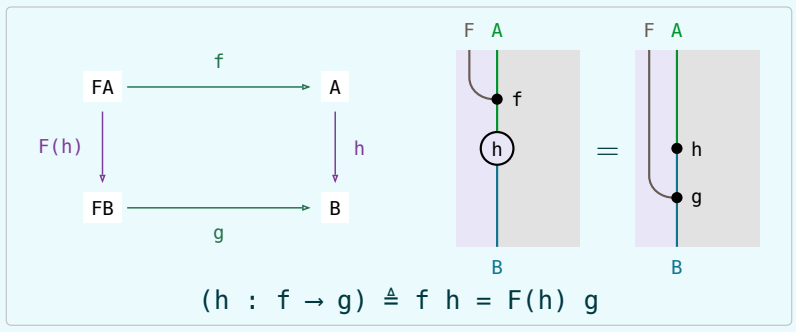
§2.4. Constructing datatypes

§2.5. Products and coproducts

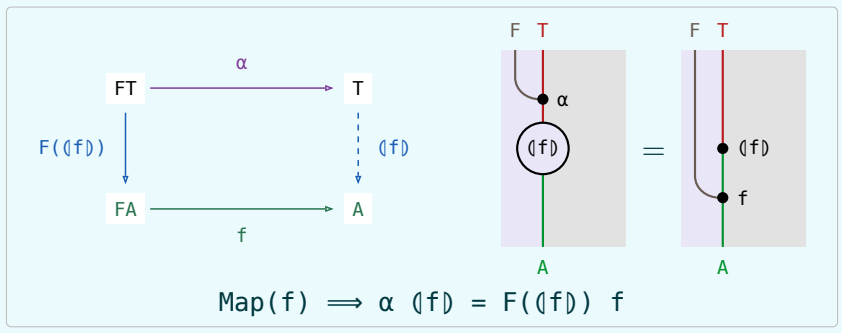
§2.6. Initial algebras

(2.6a)

The **initial algebra** `α : F(T)→T` has exactly one F-homomorphism `⦇f⦈ : T→A` to every F-algebra `f : F(A)→A`.



`(h : f → g) ≜ f h = F(h) g`



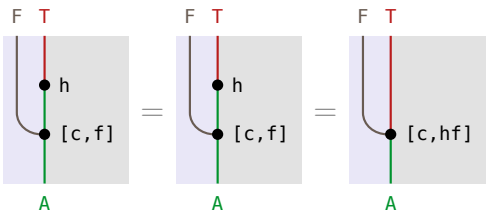
`Map(f) ⟹ α ⦇f⦈ = F(⦇f⦈) f`

(2.6b)

`α h = F(h) [c,f] ⟺ zero h = c ∧ succ h = h f`

`h` is a homomorphism from **`α = [zero,succ]`** to **`[c,f]`** exactly when **`h`** sends **`zero`** to **`c`** and **`succ`** then **`h`** equals **`h`** then **`f`**

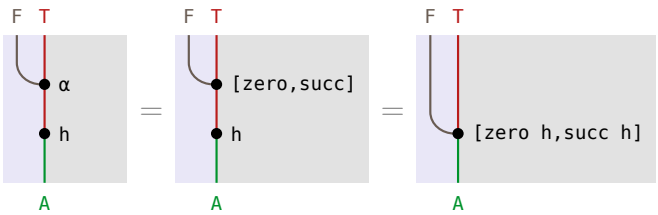
the right side



(1) (2) (3)

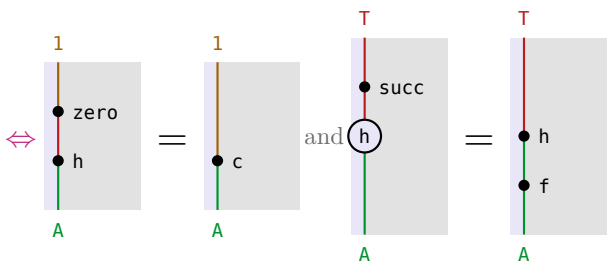
1	`F(h) [c,f]`	definition of `F`
2	`= (𝟙 + h) [c,f]`	coproduct
3	`= [c,h f]`	

the left side



(1) (2) (3)

1	`α h`	since `α = [zero,succ]`
2	`= [zero,succ] h`	coproduct
3	`= [zero h,succ h]`	cancellation, `zero` arm

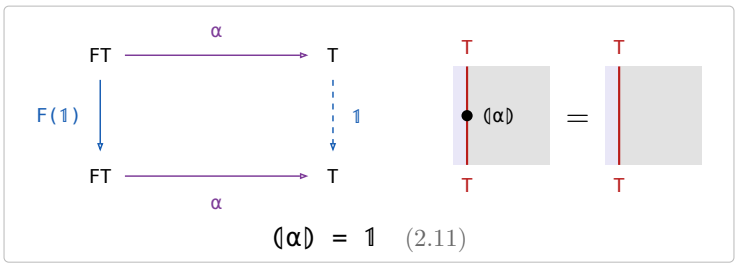


(4) (5)

4	`⟺ [zero h,succ h] = [c,h f] ⟹ zero h = c`	cancellation, `succ` arm
5	and `[zero h,succ h] = [c,h f] ⟹ succ h = h f`	

§2.6.1. Reflection

(2.6.1a)



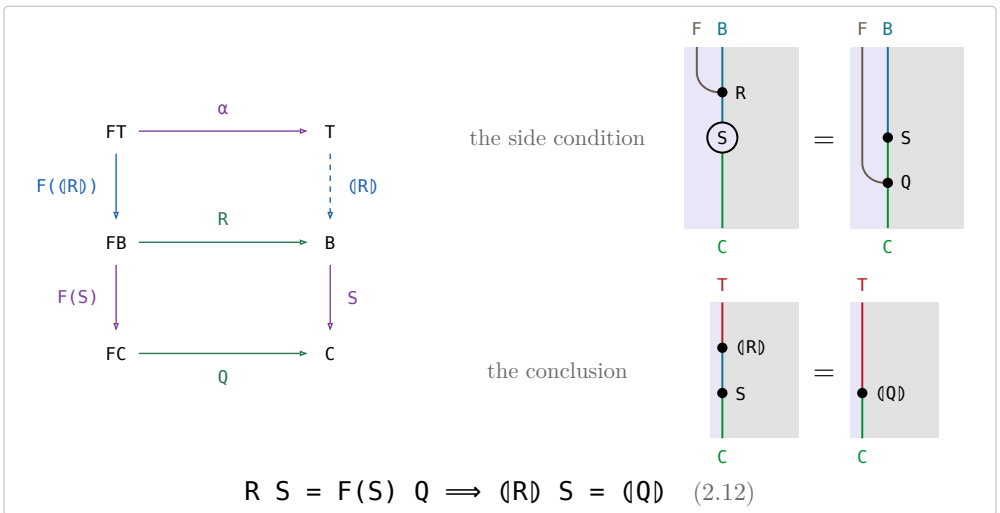
(|α|) = 𝟙 (2.11)

Taking a value apart with α and putting it straight back is doing nothing.

§2.6.2. R S = F(S) Q ⟹ (|R|) S = (|Q|) — folding with R, then applying S, is folding with Q

When S : B→C is an F-homomorphism from R : F(B)→B to Q : F(C)→C, folding with R and then applying S is folding with Q.

(2.6.2a)



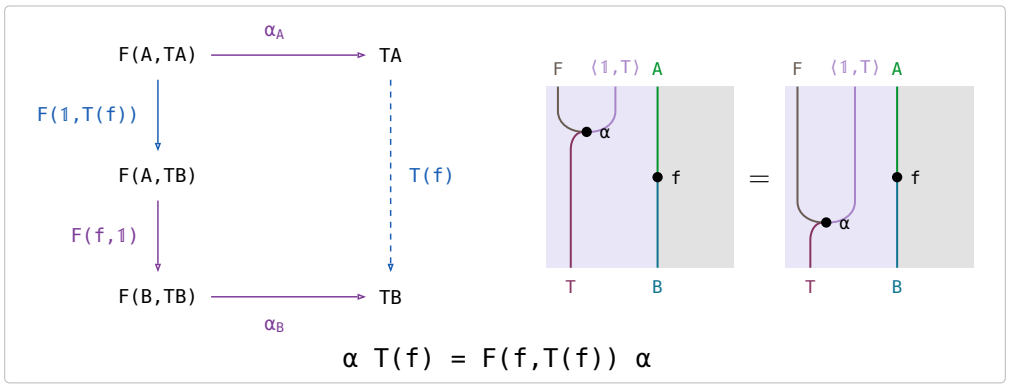
R S = F(S) Q ⟹ (|R|) S = (|Q|) (2.12)

§2.7. Type functors

(2.7a)

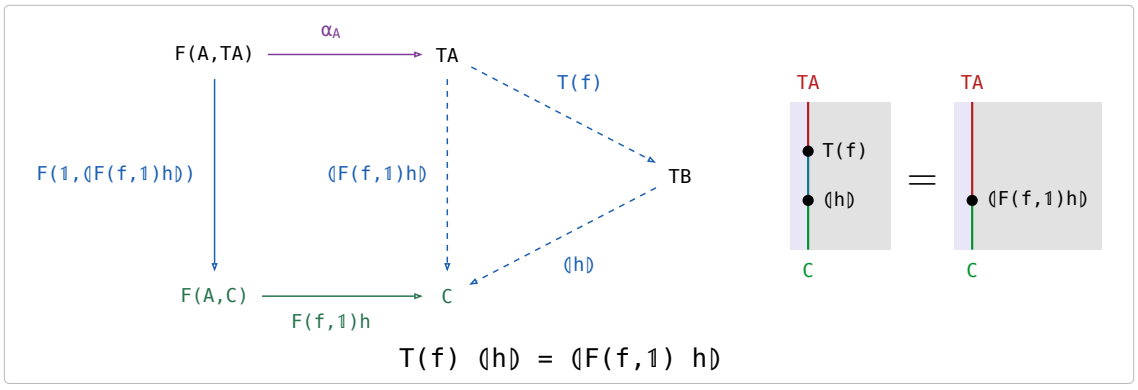
$\mathtt{T(R)} \triangleq (\!|\mathtt{F(R,\mathbb{1})}\ \alpha|\!)$ — the type functor $\mathtt{T}$ acts on a map $\mathtt{f : A \to B}$ through the initial algebras $\alpha_A : \mathtt{F(A,TA) \to TA}$.

(2.7b)



- $\mathtt{F} : \mathcal{A}\times\mathcal{A}\to\mathcal{A}$ is a bifunctor and a wire is a unary functor, so the two arguments are packed first: $\langle\mathbb{1},\mathtt{T}\rangle : \mathcal{A}\to\mathcal{A}\times\mathcal{A}$ sends $\mathtt{A}$ to $\mathtt{(A,TA)}$, and $\mathtt{F(\langle\mathbb{1},T\rangle(A))}$ is $\mathtt{F(A,TA)}$.
- The picture is three wires — $\mathtt{F}$, $\langle\mathbb{1},\mathtt{T}\rangle$, and the object — and the region between the first two is $\mathcal{A}\times\mathcal{A}$.
- α is then an ordinary natural transformation $\mathtt{F}\circ\langle\mathbb{1},\mathtt{T}\rangle\Rightarrow\mathtt{T}$: its bead eats the $\mathtt{F}$ and $\langle\mathbb{1},\mathtt{T}\rangle$ wires, and the $\mathtt{T}$ wire is born under it.
- $\mathtt{F(f,T(f))}$ costs no notation. It is the bead $\mathtt{f}$ on the object wire with $\langle\mathbb{1},\mathtt{T}\rangle$ and $\mathtt{F}$ running past: $\langle\mathbb{1},\mathtt{T}\rangle$ is what turns $\mathtt{f}$ into the pair $\mathtt{(f,T(f))}$, and $\mathtt{F}$ is what applies it.
- The law is the naturality of α, which is exactly the freedom to slide that $\mathtt{f}$ bead past it.

(2.7c)



(2.7d)

	name	law	what it says
1	functor	$\mathtt{T(\mathbb{1})} = \mathbb{1}$ and $\mathtt{T(R)\ T(S) = T(R\ S)}$	Acting by the identity changes nothing, and two actions in a row are one action.
2	type functor fusion	$\mathtt{T(f)\ (\!\|h\|\!) = (\!\|F(f,\mathbb{1})\ h\|\!)}$	A relator action followed by a fold is a single fold — the intermediate structure is never built. The side condition holds because $\mathtt{F}$ is a bifunctor — $\mathtt{F(R,\mathbb{1})F(\mathbb{1},(\!\|Q\|\!))=F(R,(\!\|Q\|\!))=F(\mathbb{1},(\!\|Q\|\!))F(R,\mathbb{1})}$.
3	naturality of α	$\mathtt{\alpha\ T(f) = F(f,T(f))\ \alpha}$	Building and then mapping is the same as mapping the parts and then building, so α is natural from $\mathtt{G(R)=F(R,T(R))}$ to $\mathtt{T}$.

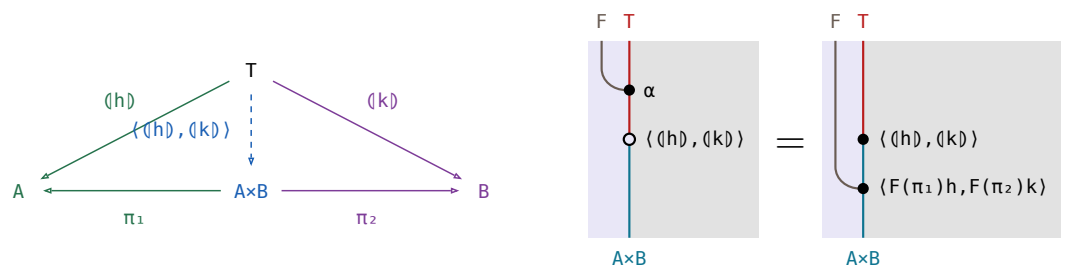
§3. Applications

§3.1. Banana-split

(3.1a)

	definition	type	note
1	`listr A::=nil|cons(A,listr A)` `list(A) = [A]`	$\mathcal{A}\to\mathcal{A}$	The cons-lists over `A`, the datatype every row below folds.
2	`sum = ⦇[zero,plus]⦈`	`[ℕ]→ℕ`	`plus(a,b)=a+b`.
3	`length = ⦇[zero,π₂ succ]⦈`	`[A]→ℕ`	`π₂` drops the head and keeps the count of the tail, `succ` adds one for the head.
4	`average ≜ ⟨sum,length⟩ div`	`[ℕ]→ℕ`	`div(m,n)=m/n`, with `div(0,0)=0` so `average` is total. Traverses the list twice.
5	banana-split law `⟨⦇h⦈,⦇k⦈⟩ = ⦇⟨F(π₁) h,F(π₂) k⟩⦈`	`T→A×B`	Any fork of folds is a single fold, hence one traversal — `F` the base functor.
6	what it reduces to `α ⟨⦇h⦈,⦇k⦈⟩ = F(⟨⦇h⦈,⦇k⦈⟩) ⟨F(π₁) h, F(π₂) k⟩`	`FT→A×B`	All that ≜ `reduce` leaves to check: the fork satisfies the defining equation.
7	the instance `⟨sum,length⟩ = ⦇[zeros,pluss]⦈`	`[ℕ]→ℕ×ℕ`	`pluss(a,(b,n))=(a+b,n+1)`, so `average` runs in one pass.
8	`preds n=[n,n−1,…,1]`	`ℕ→[ℕ]`	apply the earlier special case to write `preds` as `⦇k⦈π₁`.

(3.1b)



`⟨⦇h⦈,⦇k⦈⟩ = ⦇⟨F(π₁) h,F(π₂) k⟩⦈` banana split

§3.1.1. Fokkinga's mutual recursion theorem

- `F-Alg(𝒜)` — the algebras and their homomorphisms — has binary products whenever $\mathcal{A}$ does, and the forgetful `U : F-Alg(𝒜)→𝒜` **creates** them: the product of `h : FA→A` and `k : FB→B` is carried by `A×B`, with structure `⟨F(π₁)h,F(π₂)k⟩ : F(A×B)→A×B`.
- `π₁` and `π₂` are homomorphisms out of it, and `⟨p,q⟩ : X→A×B` is a homomorphism **iff** `p` and `q` both are — substitute the structure and compare the two forks component by component.
- The banana-split law of `⟨⦇h⦈,⦇k⦈⟩ = ⦇⟨F(π₁) h,F(π₂) k⟩⦈` IS that product's universal property read at the initial algebra: `⦇h⦈` and `⦇k⦈` are the unique homomorphisms to `(A,h)` and `(B,k)`, so their fork is the unique homomorphism into the product, which is `⦇⟨F(π₁)h,F(π₂)k⟩⦈`.
- Fokkinga's theorem is **strictly more general** and is not that product: in mutual recursion the two arrows leave `F(A×B)`, not `FA` and `FB`, so each sees BOTH components. The product is the case where they factor as `F(π₁)h` and `F(π₂)k`. What it says: **any algebra on a product carrier is folded by a fork**.

(3.1.1a)

$$\begin{tikzcd} FT \arrow[r, "\alpha"] \arrow[d, "F(\langle f,g\rangle)"'] & T \arrow[d, dashed, "f"] \\ F(A\times B) \arrow[r, "h"'] & A \end{tikzcd} \qquad \begin{tikzcd} FT \arrow[r, "\alpha"] \arrow[d, "F(\langle f,g\rangle)"'] & T \arrow[d, dashed, "g"] \\ F(A\times B) \arrow[r, "k"'] & B \end{tikzcd}$$

$$\alpha\ f = F(\langle f,g\rangle)\ h \wedge \alpha\ g = F(\langle f,g\rangle)\ k \iff \langle f,g\rangle = ⦇\langle h,k\rangle⦈$$

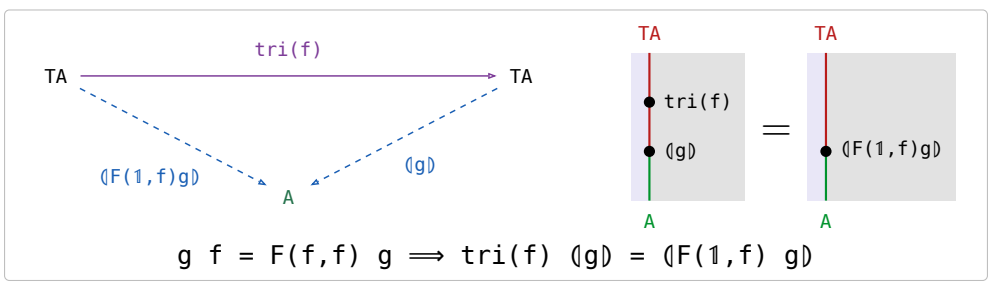
§3.2. Ruby triangles and Horner's rule

(3.2a)

	stage	definition
1	informally	`tri(f)[a₀,a₁,…,aᵢ,…,aₙ]=[a₀,f(a₁),…,fⁱ(aᵢ),…,fⁿ(aₙ)]`
2	for cons-lists	`tri(f)=⦇[nil,(𝟙×listr(f)) cons]⦈`
3	with the base functor named	`F(A,B)=1+A×B`, `α=[nil,cons]`, so `tri(f)=⦇F(𝟙,listr(f))α⦈`
4	abstractly	`F` a bifunctor with initial type `(α,T)`: `tri(f) = ⦇F(𝟙,T(f)) α⦈`

For the definition to make sense `f : A→A` is required, and then `tri(f) : TA→TA`.

(3.2b)



Horner's rule is Horner's rule, generalised from cons-lists to any initial type. Fusion reduces it to `F(𝟙,T(f))α⦇g⦈=F(𝟙,⦇g⦈)F(𝟙,f)g`.

§3.2.1. Depth of a tree

(3.2.1a)

1	the datatype	`tree A::=tip(A)|bin(tree A,tree A)`, base functor `F(A,B)=A+B×B`, initial type `([tip,bin],tree)`
2	the fold	`⦇[g,h]⦈` is the unique `f` with `f(tip(a))=g(a)` and `f(bin(x,y))=h(f(x),f(y))`
3	`tree(f)`	`tree(f)=⦇F(f,𝟙) [tip,bin]⦈`; pointwise `tree(f)(tip(a))=tip(f(a))` and `tree(f)(bin(x,y))=bin(tree(f)(x),tree(f)(y))`
4	`max`	`max=⦇[𝟙,bmax]⦈`, where `bmax(a,b)` is the larger of `a` and `b`
5	`depths`	`depths=tree(zero) tri(succ)` — replaces every tip by its depth in the tree, `zero` the constant function returning 0 and `succ` the successor
6	`depth`	`depth=depths max`

§3.3. The TeX problem

§3.4. Conditions and conditionals

§3.5. Concatenation and currying

Theorem 3.1

`If φ is natural, i.e. G(h×id)·φ=φ·(F(h)×id), then h·G(f)·φ=f ⟺ f=apply·((h·G(apply)·φ)×id)` The unique solution of a two-argument recursion `f(a,x)=h(G(f)(a,x))` built from a natural `φ` is `apply` composed with a one-argument fold, which is how `cat` — not itself a fold — is derived from one.

§4. Relations and Allegories

§4.1. Allegories

§4.2. Special properties of arrows

Proposition 4.1

`R:A→B, S:B→A, 𝟙 A⊑RS, SR⊑𝟙 B ⟹ S=R° and R is a map` A relation with a two-sided partial inverse under composition — identity below one order, identity above the other — is that inverse's converse, and R is itself a map.

§4.3. Tabular allegories

Proposition 4.2

`(f,g) a tabulation of R ⟹ k°h⊑R ⟺ ∃! map m with h=mf, k=mg` Every pair below R's tabulating pair factors, uniquely, through a single map into the tabulation's apex.

§4.4. Locally complete allegories

§4.5. Boolean allegories

§4.6. Power allegories

§5. Datatypes in Allegories

§5.1. Relators

(5.1a)

A **relator** `F : 𝒞→𝒟` is a 2-functor between allegories: it preserves `𝟙`, composition and `⊑`.

(5.1b)

	the statement
1	`Map(f) ⟹ Map(F(f))` and `Map(f) ⟹ F(f°) = F(f)°` . Lemma 5.1
2	Over a **tabular** allegory a functor is a relator ⟺ it preserves `°`. Theorem 5.1 the book's reverse direction (preserving `°` ⟹ relator) needs the source to be `Rel`: the counterexample over a tabular allegory that is not `Rel` shows it can fail
3	`F(R°) = F(R)°` for every `R`, so `F(R)°` needs no bracket.
4	Two relators agreeing on maps are equal. Corollary 5.1
5	`Coreflexive(X) ∧ Coreflexive(Y) ⟹ F(X ∩ Y) = F(X) ∩ F(Y)` Ex 5.2
6	`F(R ∩ S) ⊑ F(R) ∩ F(S)` , and strictly. Ex 5.2
7	`F(dom(R)) = dom(F(R))` Ex 5.5

The **power relator** `P` — `xs P(R) ys⟺(∀a∈xs. ∃b∈ys. a R b)∧(∀b∈ys. ∃a∈xs. a R b)` — is where the fourth is strict: for `R={(a₁,b₁),(a₂,b₂)}` and `S={(a₁,b₂),(a₂,b₁)}` the pair `({a₁,a₂},{b₁,b₂})` is in `P(R)∩P(S)`, while `R∩S=∅`.

§5.2. Relational products

(5.1)

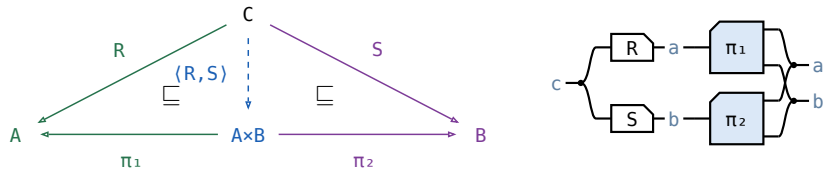
$\langle R,S\rangle \triangleq (R\ \pi_1{}^\circ) \cap (S\ \pi_2{}^\circ)$ — the **fork** of $R : C\to A$ and $S : C\to B$, with (π_1,π_2) the tabulation of $\top$.

$\langle R,S\rangle \triangleq R\pi_1{}^\circ \cap S\pi_2{}^\circ$ $c\ \langle R,S\rangle\ (a,b) \Longleftrightarrow c\ R\ a \wedge c\ S\ b$.

(5.2a)

$\langle R,S\rangle\ \pi_1 = \mathrm{dom}(S)\ R$ (5.6) $\langle R,S\rangle\ \pi_2 = \mathrm{dom}(R)\ S$ (5.7)

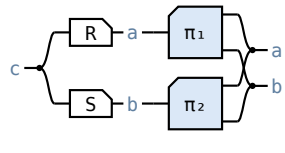
(5.2b)



A domain is coreflexive, so $\langle R,S\rangle\pi_1 \sqsubseteq R$, with equality exactly when S is entire; for maps both triangles commute and $\langle f,g\rangle$ is unique. In Rel, $c\ \langle R,S\rangle\ (a,b)$ iff $c\ R\ a$ and $c\ S\ b$ — copy c, then R on one strand and S on the other, which is $\lhd(R\otimes S)$ on the right.

No $^\circ$ survives the translation. $\pi_1 = \mathbb{1}\otimes\multimap$ discards the second component, so $\pi_1{}^\circ = \mathbb{1}\otimes\multimap$ **creates** one out of nothing, and $\cap$ is copy, run both, merge. Draw that and the created strands — the two dots with no left end, and the crossing they force — are merged against real ones, which is the monoid's unit law:

(5.2c)



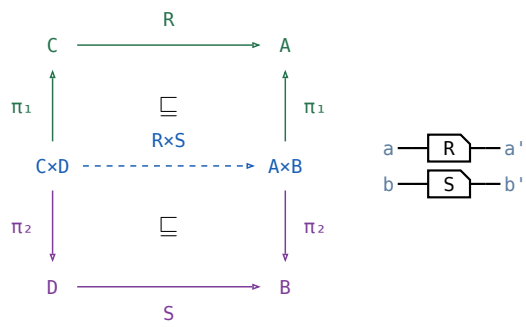
$\multimap\!\rhd = \mathbb{1}$ on each half

§5.2.1. Relational product R×S

(5.2)

$R \times S \triangleq \langle \pi_1\ R, \pi_2\ S\rangle$ — the relational product $R\times S \triangleq \langle \pi_1 R, \pi_2 S\rangle$.

(5.2.1a)

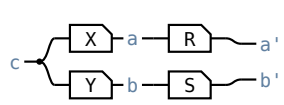


Right-then-up is $(R\times S)\pi_1$, up-then-right is $\pi_1 R$, and $(R\times S)\pi_1 \sqsubseteq \pi_1 R$, equality when S is entire. In Rel, $(c,d)\ (R\times S)\ (a,b)$ iff $c\ R\ a$ and $d\ S\ b$ — two strands side by side, no copy dot: $R\times S = R\otimes S$.

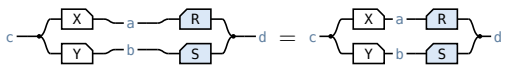
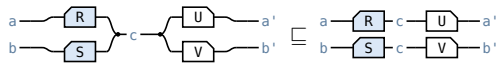
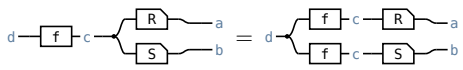
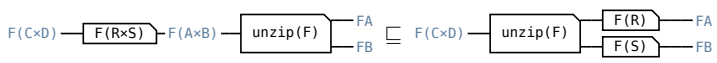
§5.2.2. Absorption

For $X : E\to C$ and $Y : E\to D$, $\langle X,Y\rangle(R\times S) = \langle XR, YS\rangle$. Both sides are this picture:

(5.3)

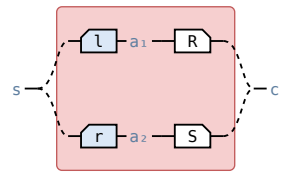
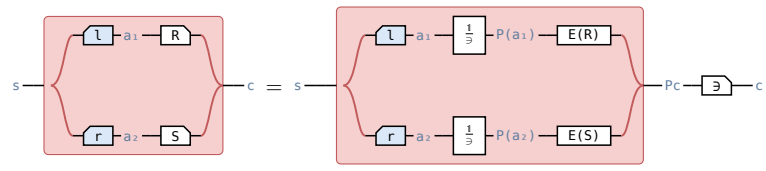
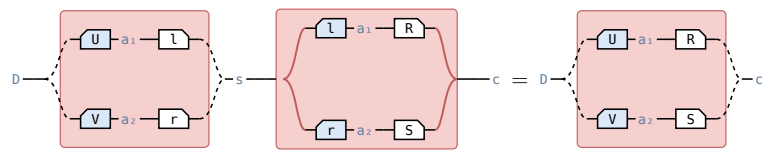


(5.2.2a)

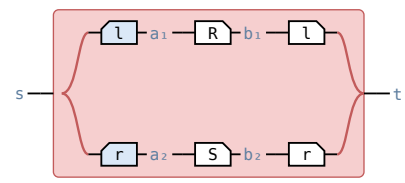
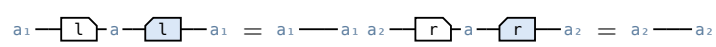
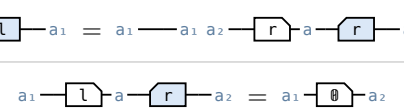
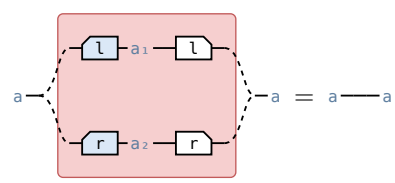
	the law	picture
1	⟨X,Y⟩ ⟨R,S⟩° = (X R°) ∩ (Y S°)	
2	⟨R,S⟩° ⟨U,V⟩ ⊑ (R° U) × (S° V)	
3	Map(f) ⟹ f ⟨R,S⟩ = ⟨f R,f S⟩ f a map; it fails for an arbitrary arrow;	
4	X ⟨R,S⟩ ⊑ ⟨X R,X S⟩ the half of the row above that holds for every X	
5	F(R × S) unzip(F) ⊑ unzip(F) (F(R) × F(S)) unzip(F)≜⟨F(π₁),F(π₂)⟩; only ⊑ for an arbitrary R, S	
6	g=curry(f)⟺(g×𝟙)eval=f reading × as the relational product, does Rel have exponentials?	

§5.3. Relational coproducts

(5.3a)

	the statement R : a₁→c, S : a₂→c	picture
1	[R,S] ≜ (l° R) ∪ (r° S) (5.9) The tape is the union — a particle entering at A+B takes exactly one branch — and the two mirrored boxes are what makes the branches disjoint.	
2	[R,S] = [R/∋, S/∋] ∋	
3	l [R,S] = R , r [R,S] = S l T = R ∧ r T = S ⟹ T = [R,S] ,	
4	U : a₁→D, V : a₂→D [U,V]° [R,S] = (U° R) ∪ (V° S) (5.11)	
5	Z : c→D [R,S] Z = [R Z,S Z]	

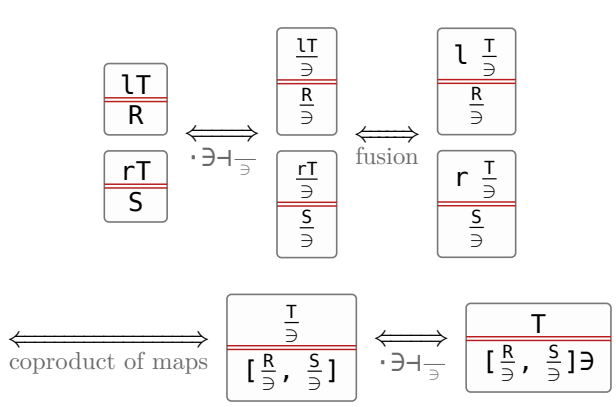
(5.3b)

	the statement R : a₁→b₁, S : a₂→b₂	picture
1	R + S ≜ [R l,S r] (5.10)	
2	l l° = 𝟙 , r r° = 𝟙 Ex 5.12	
3	l r° = 𝟘 , r l° = 𝟘 Ex 5.12	
4	(l° l) ∪ (r° r) = 𝟙 Ex 5.12	

§5.3.1. [R,S] = [R/∋, S/∋] ∋ — a case of relations is the case of their transposes, then ∋

The universal-property row is not free: l,r were only ever asked to be a coproduct of **maps**. They stay one once every arrow is allowed because $\frac{\ }{\ni}$ sends an arrow A→C to a map A→PC reversibly, so the map coproduct can be applied underneath it. For any T : A+B→C,

(5.3.1a)

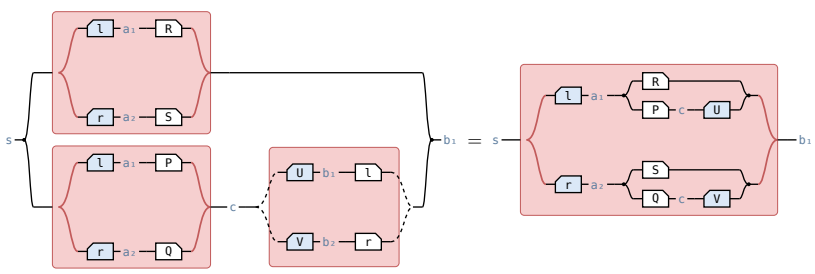


(5.3.1b)

$$\begin{tikzcd}
 & & A \arrow[dl, "l"'] \arrow[dr, "\frac{R}{\ni}"] \arrow[drr, bend left, "R"] & & \\
A+B \arrow[rrr, dashed, "{[\frac{R}{\ni}, \frac{S}{\ni}]}"] & & & PC \arrow[r, "\ni"] & C \\
 & & B \arrow[ul, "r"] \arrow[ur, "\frac{S}{\ni}"'] \arrow[urr, bend right, "S"'] & &
\end{tikzcd}$$

Nothing here holds only up to ⊑: every triangle commutes on the nose, which is the difference from the fork above. The border spells `[R,S]=[`$\frac{R}{\ni}$`, `$\frac{S}{\ni}$`]∋`, and pushing ∋ into the union that `[·,·]` on maps already is turns that back into the definition, `(l°`$\frac{R}{\ni}$` ∪r°`$\frac{S}{\ni}$`)∋=l°R ∪ r°S`.

(5.3.1c)

	the law	picture
1	prove (5.11), and say why duality does not carry it over from the product law	
2	`(R + S) ∩ ([P,Q] [U,V]°) = (R ∩ (P U°)) + (S ∩ (Q V°))` `[P,Q][U,V]°` is a full 2×2 of composites; `R+S` is diagonal, so the meet cuts the two off-diagonal branches	

§5.4. The power relator P(R)

(5.4a)

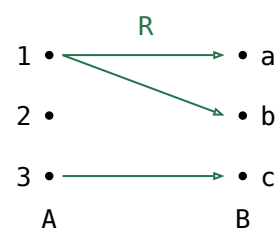
P(R) ≜ (∈ \ (R ∈)) ∩ ((∋ R) / ∋) — the power relator P(R) : PA→PB, R lifted to sets.

xs P(R) ys⟺(∀a∈xs. ∃b∈ys. a R b)∧(∀b∈ys. ∃a∈xs. a R b)

(5.4b)

	in Rel	in words	im(xs)	partner
1 (∋R)/∋	∀y∈ys. ∃x∈xs. x R y	∀y. some xs R y	ys⊆im(xs)	every y has a partner
2 (∋/(∋R))°	∀y. (∃x∈xs. x R y)→y∈ys	every y with some xs R y is in ys	ys⊇im(xs)	ys leaves out no partner
3 E(R)	ys={y|∃x∈xs. x R y}	ys = every y with some xs R y	ys=im(xs)	every y has a partner, and none is left out
4 ((∋R°)/∋)°	∀x∈xs. ∃y∈ys. x R y	∀x. x R some ys	—	every x has a partner
5 P(R)	P(R)(xs,ys) ⟺ (∀y. y ∈ ys ⟹ (∃x. x ∈ xs ∧ R(x,y))) ∧ (∀x. x ∈ xs ⟹ (∃y. y ∈ ys ∧ R(x,y)))	∀y. some xs R y and ∀x. x R some ys	—	every x and every y has a partner

(5.4c)



	xs	(∋R)/∋	(∋/(∋R))°	E(R)	((∋R°)/∋)°	P(R)
1	∅	∅	all 8 subsets of abc	∅	all 8 subsets of abc	∅
2	{1}	∅, a, b, ab	ab, abc	ab	a, b, ab, ac, bc, abc	a, b, ab
3	{2}	∅	all 8 subsets of abc	∅	none	none
4	{3}	∅, c	c, ac, bc, abc	c	c, ac, bc, abc	c
5	{1,2}	∅, a, b, ab	ab, abc	ab	none	none
6	{1,3}	all 8 subsets of abc	abc	abc	ac, bc, abc	ac, bc, abc
7	{2,3}	∅, c	c, ac, bc, abc	c	none	none
8	{1,2,3}	all 8 subsets of abc	abc	abc	none	none

(5.4d)

	law	the reading
1	X ⊑ P(R) ⟺ X ∋ ⊑ ∋ R ∧ X° ∋ ⊑ ∋ R°	One containment, and the same one at R° — which is the definition read off the two divisions. Hence P(R°)=P(R)°, and R⊑S⟹P(R)⊑P(S) .
2	P(𝟙) = 𝟙	The straightness axiom verbatim: extensionality **is** P's unit law.
3	Map(f) ⟹ P(f) = E(f)	In Rel, xs P(f) ys⟺ys={f(a)|a∈xs}. The half at f° says every a∈xs has its f(a) on ys; f has just the one image per a, so that already says ys contains everything xs reaches, which is the fraction's second half. For a map the two definitions coincide.
4	P(R S) = P(R) P(S)	⊒ is the division cancellation laws. ⊑ is the one law in this section that is not a calculation: it needs a tabulation of P(RS).

§5.4.1. Type relators

(5.4.1a)

T(R) = ⦇F(R,𝟙) α⦈ — the type functor T acts on an arrow R : A→B by rebuilding with α, applying R to the parameter.

(5.4.1b)

	name	law	what it says
1	the defining equation	T(R) = ⦇F(R,𝟙) α⦈	Rebuild the structure with α, applying R to the parameter on the way.
2	type relator	T(R)° = T(R°)	A datatype acts on relations, not only on maps — the map of the converse is the converse of the map.

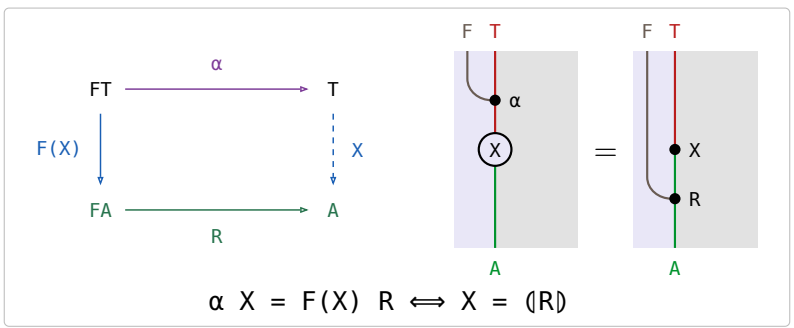
§5.5. Relational catamorphisms

(5.5a)

The initial algebra α : F(T)→T among the maps is also initial in the allegory:

§5.5.1. The defining equation

(5.5.1a)



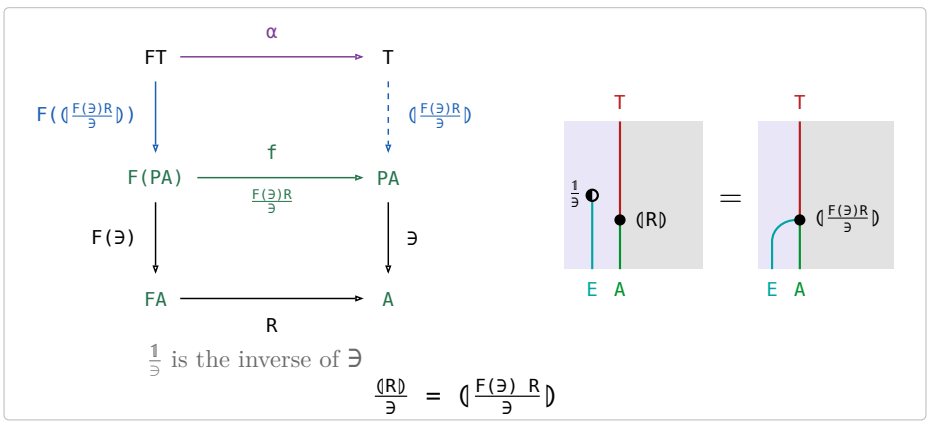
An F-algebra is a **weakened** natural transformation. A transformation F⇒𝟙 on $\mathcal{A}$ would need a component FX→X at every object and a commuting square at every arrow, but F-Algebra only need it works on T and A.

(5.5.1b)

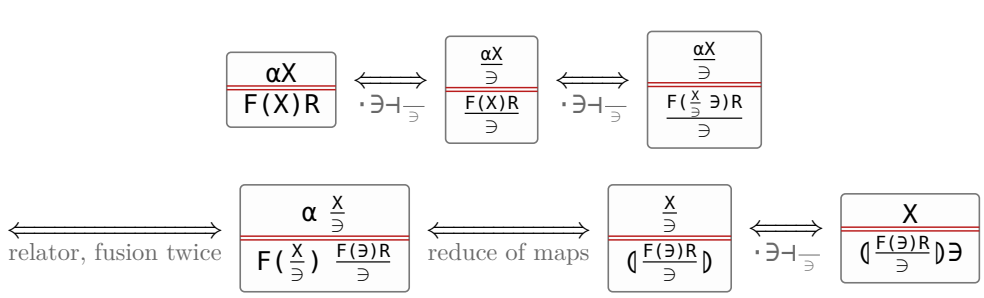
⦇[c,f]⦈=reduce(c,f) catamorphism: arrows of the form ⦇f⦈ are called catamorphisms, and these two examples reveal the notion to be a familiar idea in abstract clothing		
part	**Nat**	**[A]**
datatype	Nat::=zero|succ Nat	[A]::=nil|cons(A,[A])
base functor F	F(X)=1+X	F(X)=1+A×X
initial algebra α	α=[zero,succ] : 1+Nat→Nat	α=[nil,cons] : 1+A×[A]→[A]
the fold, pointwise	⦇[c,f]⦈(zero)=c ⦇[c,f]⦈(succ(n))=f(⦇[c,f]⦈(n))	⦇[c,f]⦈(nil)=c ⦇[c,f]⦈(cons(a,x))=f(a,⦇[c,f]⦈(x))

§5.5.2. ⦇R⦈ = ⦇$\frac{F(\ni)\ R}{\ni}$⦈ ∋ — folding with R is a fold of maps into sets, then ∋

(5.5.2a)



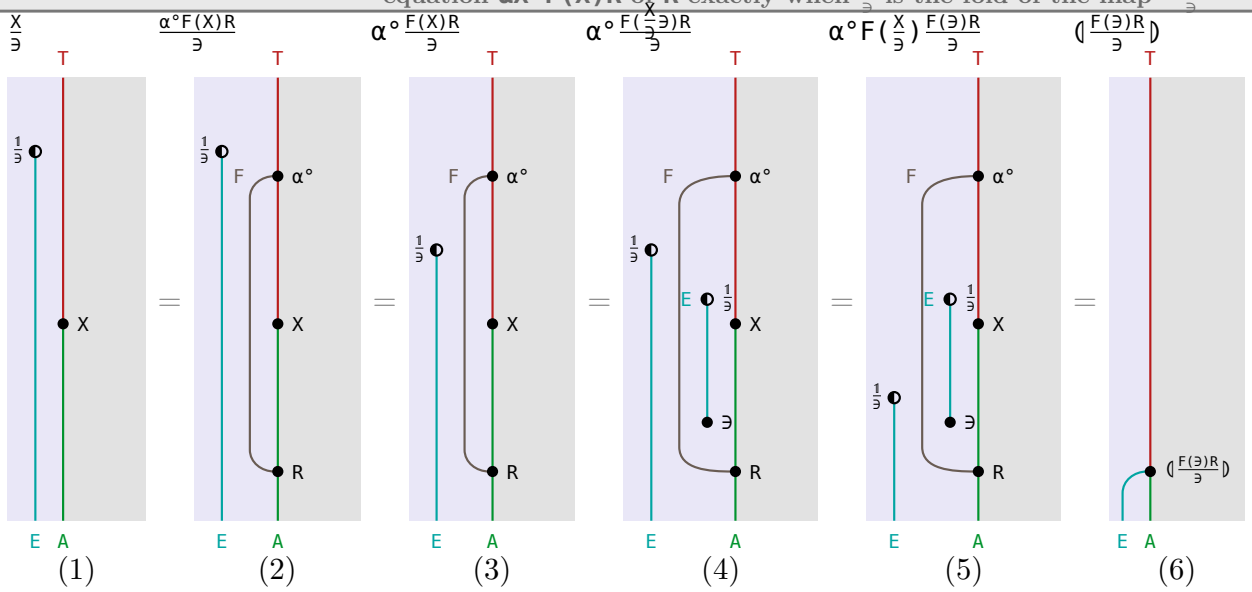
(5.12)



the last two rows at X:=⦇R⦈:

$$\alpha\ X = F(X)\ R \iff \frac{X}{\ni} = (\!|\frac{F(\ni)\ R}{\ni}|\!) \qquad (5.5.2b)$$

in a tabular allegory, with F a relator and $\alpha : F(T) \to T$ its initial algebra: a relation X out of T satisfies the fold equation $\alpha X = F(X)R$ of R exactly when $\frac{X}{\ni}$ is the fold of the map $\frac{F(\ni)R}{\ni}$



1	$\frac{X}{\ni}$	α iso; $\alpha X = F(X)R$
2	$= \frac{\alpha^\circ\ F(X)\ R}{\ni}$	α° is a map because α is an iso, so it leaves Λ: $\frac{\alpha^\circ F(X)R}{\ni} = \alpha^\circ \frac{F(X)R}{\ni}$ — $Map(f)$ $\Longrightarrow \frac{f\ R}{\ni} = f\ \frac{R}{\ni}$
3	$= \alpha^\circ\ \frac{F(X)\ R}{\ni}$	$X = \frac{X}{\ni}\ni$ — $\frac{R}{\ni}\ \ni = R$
4	$= \alpha^\circ\ \frac{F(\frac{X}{\ni}\ \ni)\ R}{\ni}$	$F(\frac{X}{\ni}\ni) = F(\frac{X}{\ni})F(\ni)$ as F is a functor, and $F(\frac{X}{\ni})$ is a map (a relator sends maps to maps), so it leaves Λ — $\triangleq$ relator, $Map(f)$ $\Longrightarrow \frac{f\ R}{\ni} = f\ \frac{R}{\ni}$
5	$= \alpha^\circ\ F(\frac{X}{\ni})\ \frac{F(\ni)\ R}{\ni}$	fold uniqueness, $\frac{F(\ni)R}{\ni}$ being a map — $(h : f \to g) \triangleq f\ h = F(h)\ g$, $Map(f)$ $\Longrightarrow \alpha\ (\!|f|\!) = F((\!|f|\!))\ f$
6	$= (\!|\frac{F(\ni)\ R}{\ni}|\!)$	

§5.6. Combinatorial functions

(5.6a)

	definition	type	note
1	`[A]::=nil|cons(A,[A])` `list(A) = [A]`	$\mathcal{A}\to\mathcal{A}$	The list type, under the short name it keeps.
2	`list(R) = ⦇[nil,(R × 𝟙) cons]⦈`	`[A]→[B]`	The relator's action on `R : A→B`: one `R` per element, the shape untouched.
3	`subseq = ⦇[nil,cons ∪ π₂]⦈`	`[A]→[A]`	`xs subseq ys`: `ys` is `xs` with elements dropped — `cons` keeps the head, `π₂` drops it.
4	`prefix = ⦇[nil,cons ∪ ⊸ nil]⦈` `prefix = cat° π₁ =init*`	`[A]→[A]`	`ys` is an initial segment of `xs`; the first `nil` is where it stops early. `init≜snoc° π₁`.
5	`suffix = cat° π₂ =tail*`	`[A]→[A]`	The dual, `tail≜cons° π₂`; as a reduce it needs snoc-lists.
6	`partition = concat°`	`[A]→[[A]]`	This `cat` is restricted to `[A]⁺×[A]→[A]`, so `ys` is a list of non-empty segments of `xs`.
7	`concat = ⦇[nil,cat]⦈`	`[[A]]→[A]`	Joins the segments back up, which is why its converse splits a list.
8	`inits`	`[A]→[[A]]`	Implements $\frac{\texttt{prefix}}{\ni}$, listing the prefixes by increasing length.
9	`tails`	`[A]→[[A]]`	Implements $\frac{\texttt{suffix}}{\ni}$ by decreasing length — the opposite order.

10	`cpp`, `cpl`, `cpr`	`[A]×[B]→[A×B]`
		`[A]×B→[A×B]`
		`A×[B]→[A×B]`
			Every pair `(a,b)` with `a` from the first list and `b` from the second; `cpl`, `cpr` keep one side a single element.

11	`cpp setify = (setify × setify)` $\frac{\ni\times\ni}{\ni}$ `cpl setify = (setify × 𝟙)` $\frac{\ni\times 1}{\ni}$ `cpr setify = (𝟙 × setify)` $\frac{1\times\ni}{\ni}$	`[A]×[B]→P(A×B)`	Forgetting the order with `setify`, `cpp` is the transpose of `∋×∋`.
12	`cp(F)` $\triangleq \frac{F(\ni)}{\ni}$	`F(PA)→PFA`	`cp(F)`, the transpose of a relator's action on `∋`; `cpp` implements it at `F(A)=A×A`.

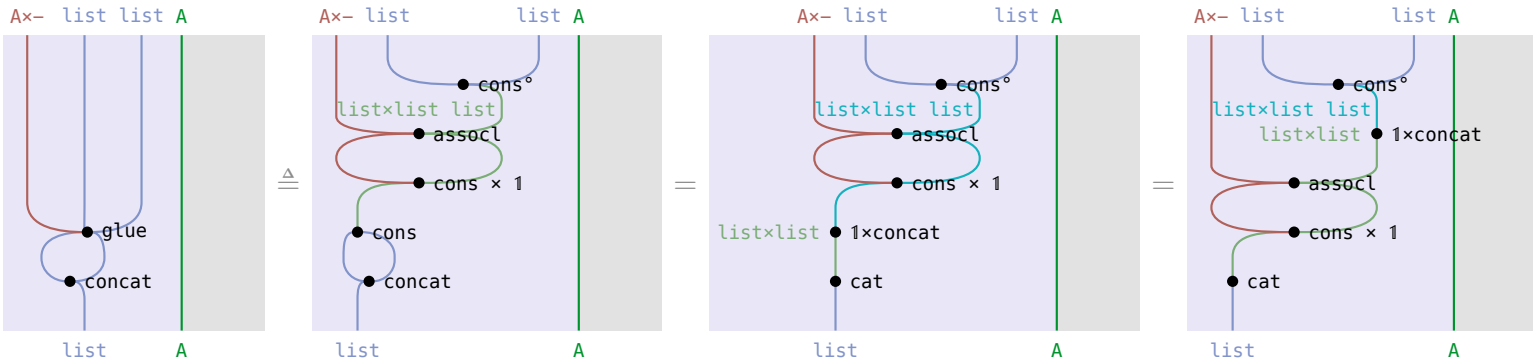
13	`cp(list) = ⦇`$\frac{[nil,(\ni\times\ni)\ cons]}{\ni}$`⦈`
	$\frac{[nil,(\ni\times\ni)\ cons]}{\ni}$ `= [nil` $\frac{1}{\ni}$`,` $\frac{\ni\times\ni}{\ni}$
	`E(cons)]`
		`list(PA)→P(list(A))`	`cp(list)`: (5.12) `⦇R⦈ = ⦇`$\frac{F(\ni)\ R}{\ni}$`⦈ ∋` at `list(∋)=⦇[nil, (∋×𝟙)cons]⦈`, then the algebra expanded.

14	`cplist = ⦇[nil wrap,cpp list(cons)]⦈`	`[[A]]→[[A]]`	`cp(list)` with every set a list: `cpp` in place of $\frac{\ni\times\ni}{\ni}$.

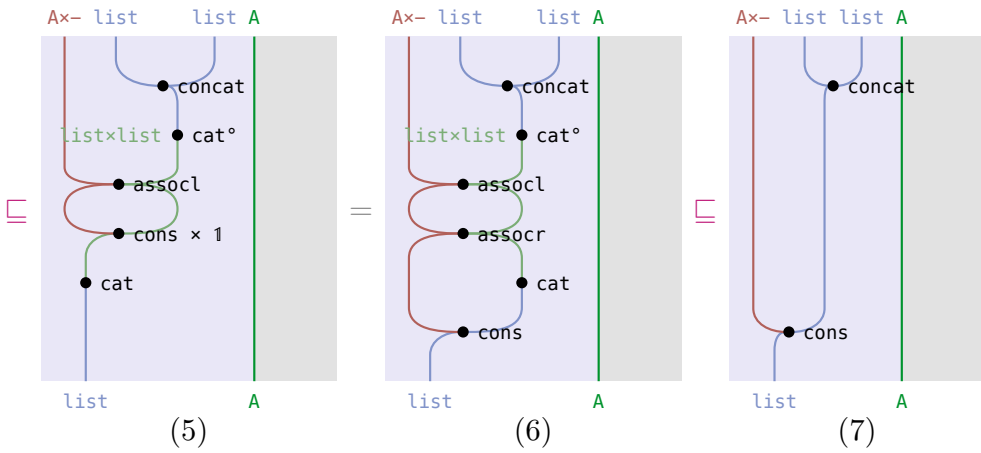
glue concat ⊑ (𝟙 × concat) cons (5.6b)

putting **a** on the front of the first segment and flattening (**glue concat**) gives at most flattening the segments and putting **a** on the front of the result (**(𝟙×concat) cons**)

the starting composite



1	glue concat	definition of glue
2	≜ (𝟙 × cons°) assocl cons × 𝟙 cons concat	since concat=⦇[nil,cat]⦈
3	= (𝟙 × cons°) assocl cons × 𝟙 (𝟙 × concat) cat	naturality of assocl
4	= (𝟙 × (cons° (𝟙 × concat))) assocl cons × 𝟙 cat	since concat=⦇[nil,cat]⦈



5	⊑ (𝟙 × concat) (𝟙 × cat°) assocl cons × 𝟙 cat	since (cons×𝟙) cat=assocr (𝟙×cat) cons
6	= (𝟙 × concat) (𝟙 × cat°) assocl assocr (𝟙 × cat) cons	since assocl assocr=𝟙 and cat° cat⊑𝟙
7	⊑ (𝟙 × concat) cons	

§5.6.1. $\frac{\text{subseq}}{\ni}$ = ⦇[nil $\frac{𝟙}{\ni}$,⟨$\frac{𝟙 \times \ni}{\ni}$ E(cons),π₂⟩ cup]⦈ — the set of subsequences is a fold of maps into sets

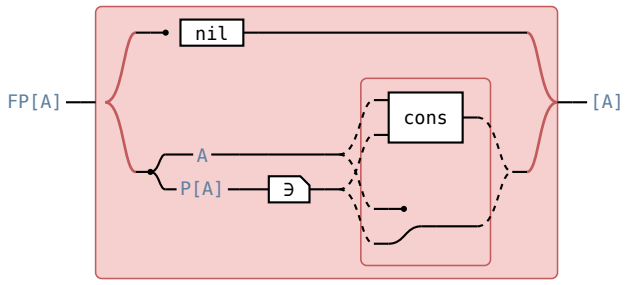
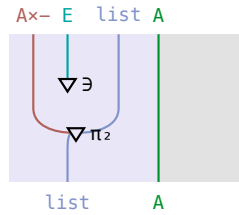
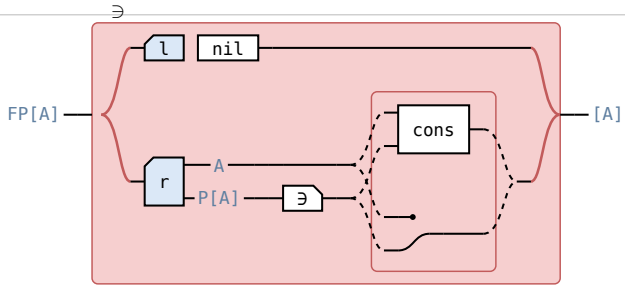
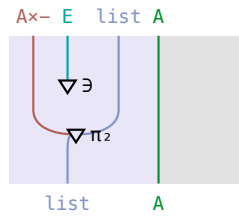
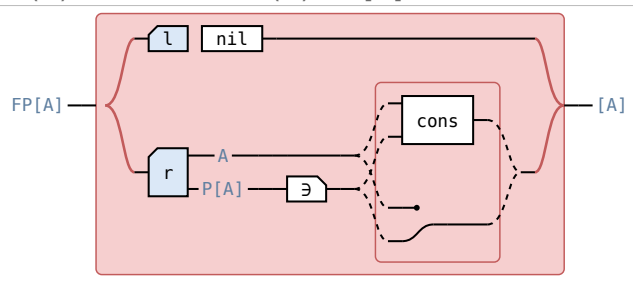
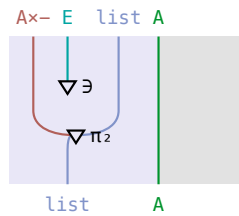
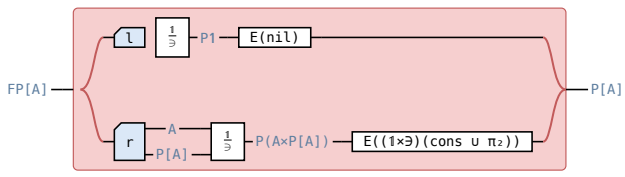
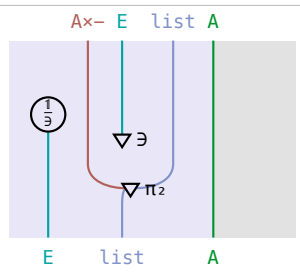
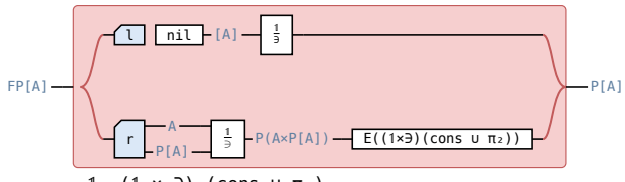
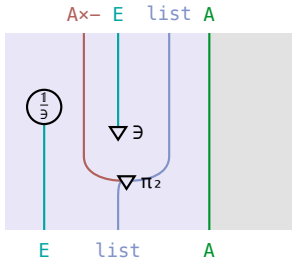
(5.6.1a)

cup ≜ $\frac{(\pi_1\ \ni)\ \cup\ (\pi_2\ \ni)}{\ni}$ — cup unions two sets.

cup≜ $\frac{\pi_1\ni\ \cup\ \pi_2\ni}{\ni}$: PA×PA→PA

$$\frac{\texttt{F(∋) [nil,cons ∪ π₂]}}{\ni} = \texttt{[nil}\ \tfrac{\mathbb{1}}{\ni}\texttt{,}\ \frac{\texttt{(𝟙 × ∋) (cons ∪ π₂)}}{\ni}\texttt{]} \tag{5.6.1b}$$

the set of lists the algebra builds is, from nothing, just **nil**, and from a head and a set of tails, every tail in the set with the head put on or left off — (5.12) $(\!|R|\!) = (\!|\frac{F(\ni)\ R}{\ni}|\!)\ \ni$ at **subseq**=$(\!|$[nil,cons ∪ π₂]$|\!)$, **subseq** = $(\!|$[nil,cons ∪ π₂]$|\!)$.

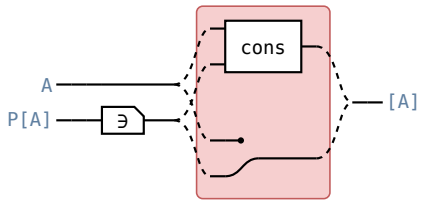
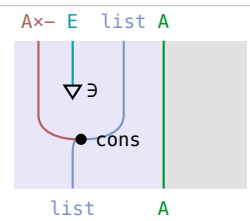
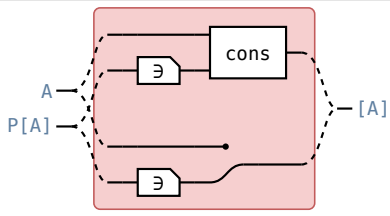
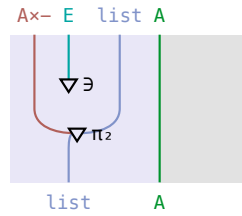
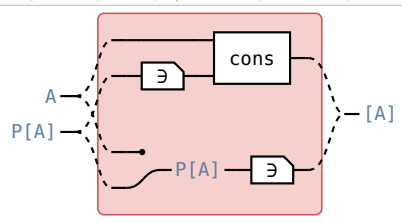
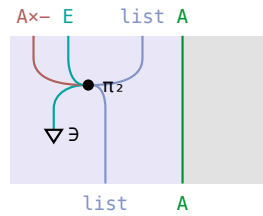
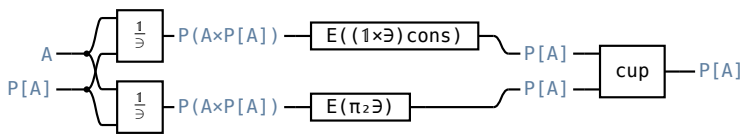
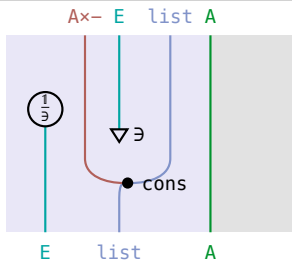
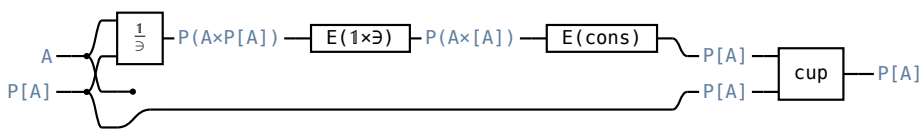
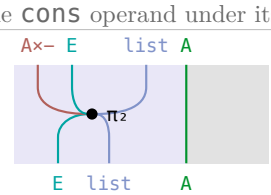
circuit — the fork is F([A])=1+A×[A]: nil above, the pair below	**Hinze–Marsden**
 $\frac{\texttt{F(∋) [nil,cons ∪ π₂]}}{\ni}$	 the π₂ operand of cons ∪ π₂ under the 𝟙×∋ summand of F(∋), i.e. (𝟙×∋)π₂
=  $\frac{\texttt{(𝟙 + (𝟙 × ∋)) [nil,cons ∪ π₂]}}{\ni}$ F(X)=1+A×X — list(A) = [A]	 the same operand under 𝟙+𝟙×∋, whose 𝟙×∋ summand it sits in
=  $\frac{\texttt{[nil,(𝟙 × ∋) (cons ∪ π₂)]}}{\ni}$ R+S≜[Rl,Sr], l[R,S]=R, r[R,S]=S — [R,S] ≜ (l° R) ∪ (r° S) , l [R,S] = R , r [R,S] = S , l T = R ∧ r T = S ⟹ T = [R,S]	 the π₂ operand of the second arm (𝟙×∋)(cons ∪ π₂)
=  $[\frac{\texttt{nil}}{\ni}, \frac{\texttt{(𝟙 × ∋) (cons ∪ π₂)}}{\ni}]$ [R,S] = $[\frac{R}{\ni}, \frac{S}{\ni}]$ ∋ at T:=[nil,(𝟙×∋)(cons ∪ π₂)]	 the π₂ operand under its $\frac{\mathbb{1}}{\ni}$, the arm (𝟙 × ∋) π₂ = π₂ ∋ rewrites
=  $[\texttt{nil}\ \frac{\mathbb{1}}{\ni}, \frac{\texttt{(𝟙 × ∋) (cons ∪ π₂)}}{\ni}]$ Map(f) ∧ 𝟙□ = ∋□ ⟹ $\frac{f}{\ni}$ = f $\frac{\mathbb{1}}{\ni}$, $\frac{f}{\ni}$ =f $\frac{\mathbb{1}}{\ni}$ for f a map, at f:=nil	 the same operand; the two rows differ only in the nil arm

(5.6.1c)

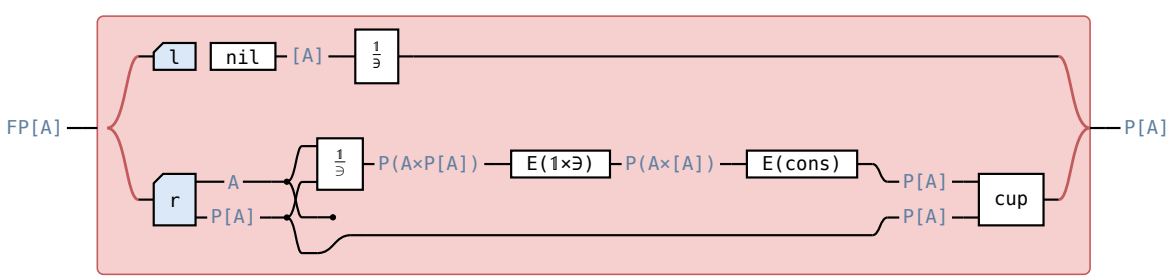
$$\begin{tikzcd} A\times P[A] \arrow[r, "\pi_2"] \arrow[d, "\mathbb{1}\times\ni"'] & P[A] \arrow[d, "\ni"] \\ A\times[A] \arrow[r, "\pi_2"'] & {[A]} \end{tikzcd}$$

$$\frac{(\mathbb{1} \times \ni)\ (\mathtt{cons} \cup \pi_2)}{\ni} = \langle \frac{\mathbb{1} \times \ni}{\ni}\ \mathtt{E(cons)}, \pi_2 \rangle\ \mathtt{cup} \tag{5.6.1d}$$

power transpose of join: the power transpose of the join of two relations is $\langle \frac{R}{\ni}, \frac{S}{\ni} \rangle$ **cup**, where **cup** is the function that returns the union of two sets

circuit	Hinze–Marsden
 $\frac{(\mathbb{1} \times \ni)\ (\mathtt{cons} \cup \pi_2)}{\ni}$ $\frac{\mathtt{F}(\ni)\ [\mathtt{nil}, \mathtt{cons} \cup \pi_2]}{\ni} = [\mathtt{nil}\ \frac{1}{\ni}, \frac{(\mathbb{1} \times \ni)\ (\mathtt{cons} \cup \pi_2)}{\ni}]$'s second branch	 the cons operand of $\mathtt{cons} \cup \pi_2$
=  $\frac{((\mathbb{1} \times \ni)\ \mathtt{cons}) \cup ((\mathbb{1} \times \ni)\ \pi_2)}{\ni}$ $\mathtt{T}(X_1 \cup X_2) = \mathtt{T}X_1 \cup \mathtt{T}X_2$ — $(S \cup T)\ R = (S\ R) \cup (T\ R)$, $R\ (S \cup T) = (R\ S) \cup (R\ T)$, $R \cap (S \cup T) = R \cap S \cup R \cap T$, $(R \cup S)^\circ = S^\circ \cup R^\circ$	 the π_2 operand of $(\mathbb{1}\times\ni)\mathtt{cons} \cup (\mathbb{1}\times\ni)\pi_2$
=  $\frac{((\mathbb{1} \times \ni)\ \mathtt{cons}) \cup (\pi_2\ \ni)}{\ni}$ $(\mathbb{1}\times\ni)\pi_2 = \pi_2\ni$ — $(R \times S)\ \pi_1 \sqsubseteq \pi_1\ R \wedge (R \times S)\ \pi_2 \sqsubseteq \pi_2\ S$ at π_2, an equality because $\mathbb{1}$ is entire	 the π_2 operand of $(\mathbb{1}\times\ni)\mathtt{cons} \cup \pi_2\ni$
=  $\langle \frac{(\mathbb{1} \times \ni)\ \mathtt{cons}}{\ni}, \frac{\pi_2\ \ni}{\ni} \rangle\ \mathtt{cup}$ $\frac{R \cup S}{\ni} = \langle \frac{R}{\ni}, \frac{S}{\ni} \rangle\ \mathtt{cup}$ — $\triangleq \mathtt{cup}$	 the cons operand under its $\frac{1}{\ni}$
=  $\langle \frac{\mathbb{1} \times \ni}{\ni}\ \mathtt{E(cons)}, \pi_2 \rangle\ \mathtt{cup}$ $\frac{S}{\ni}\ \mathtt{E}(R) = \frac{S\ R}{\ni}$, absorption $\frac{S}{\ni}\ \mathtt{E}(R) = \frac{SR}{\ni}$ at $S := \mathbb{1}\times\ni$, $R := \mathtt{cons}$; fusion and $\frac{\ni}{\ni} = \mathbb{1}$ on the π_2 operand	 the π_2 operand, bare π_2

(5.6.1e)



$$[\frac{\mathtt{nil}}{\ni}, \langle \frac{\mathbb{1}\times\ni}{\ni}\ \mathtt{E(cons)}, \pi_2 \rangle\ \mathtt{cup}]$$

which writes $\mathtt{Pcons}$; $\mathtt{cons}$ is a map, and there $\mathtt{P(cons)} = \mathtt{E(cons)}$ — $\mathtt{Map(f)}$

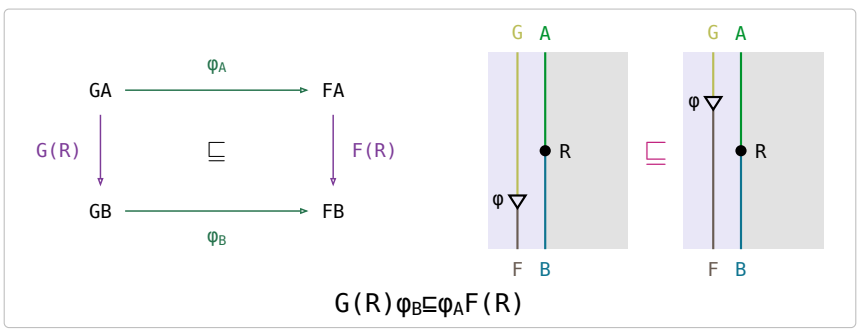
$$\Longrightarrow \mathtt{P(f)} = \mathtt{E(f)}\ .$$

§5.7. Lax natural transformations (LaT)

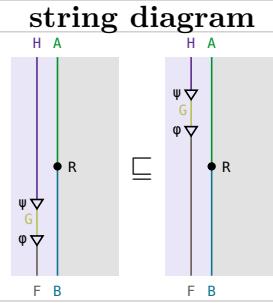
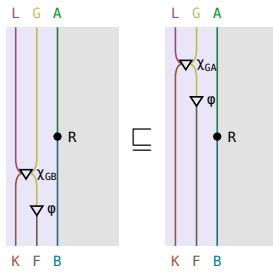
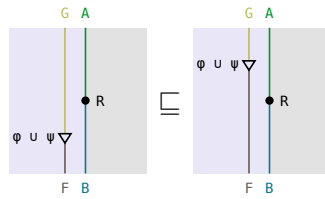
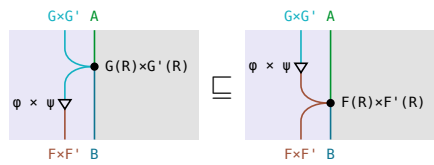
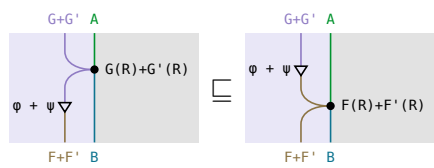
(5.7a)

$\varphi : G \Rightarrow F$ is **lax at** $R : A \to B$ when $G(R)\varphi_B \sqsubseteq \varphi_A F(R)$; it is a **lax natural transformation** (LaT) when lax at every R.

(5.7b)



(5.7c)

	closed under	commutative diagram	string diagram
1	composition $\psi\varphi$	$\begin{tikzcd} HA \arrow[r, "\psi_A"] \arrow[d, "H(R)"'] \arrow[dr, phantom, "\sqsubseteq"] & GA \arrow[r, "\varphi_A"] \arrow[d, "G(R)"'] \arrow[dr, phantom, "\sqsubseteq"] & FA \arrow[d, "F(R)"] \\ HB \arrow[r, "\psi_B"'] & GB \arrow[r, "\varphi_B"'] & FB \end{tikzcd}$ $H(R)\psi_B \sqsubseteq \psi_A G(R)$ and $G(R)\varphi_B \sqsubseteq \varphi_A F(R)$ give $H(R)(\psi_B\varphi_B) \sqsubseteq (\psi_A\varphi_A)F(R)$	
2	horizontal composition $\chi\circ\varphi$	$\begin{tikzcd} LGA \arrow[r, "\chi_{GA}K(\varphi_A)"] \arrow[d, "L(G(R))"'] \arrow[dr, phantom, "\sqsubseteq"] & KFA \arrow[d, "K(F(R))"] \\ LGB \arrow[r, "\chi_{GB}K(\varphi_B)"'] & KFB \end{tikzcd}$ $\varphi : G \Rightarrow F$ with $G,F : \mathcal{C}\to\mathcal{D}$ and $\chi : L \Rightarrow K$ with $L,K : \mathcal{D}\to\mathcal{E}$ give $\chi\circ\varphi : L\circ G \Rightarrow K\circ F$, the family $A \mapsto \chi_{GA}K(\varphi_A)$ $L(G(R))\chi_{GB} \sqsubseteq \chi_{GA}K(G(R))$ is χ lax at $G(R)$; then $K(G(R)\varphi_B) \sqsubseteq K(\varphi_A F(R))$ is K applied to φ's own inequation; $\chi := \mathbb{1}_K$ gives $K(\varphi) : K\circ G \Rightarrow K\circ F$, K composed on the outside; $\varphi := \mathbb{1}_G$ gives $\chi G : L\circ G \Rightarrow K\circ G$, G composed on the inside two candidates, $\chi_{GA}K(\varphi_A)$ and $L(\varphi_A)\chi_{FA}$, ordered by $\sqsubseteq$; this row is the first	
3	union $\varphi \cup \psi$	$\begin{tikzcd} GA \arrow[r, "\varphi_A"] \arrow[d, "G(R)"'] \arrow[dr, phantom, "\sqsubseteq"] & FA \arrow[d, "F(R)"] \\ GB \arrow[r, "\varphi_B"'] & FB \end{tikzcd} \cup \begin{tikzcd} GA \arrow[r, "\psi_A"] \arrow[d, "G(R)"'] \arrow[dr, phantom, "\sqsubseteq"] & FA \arrow[d, "F(R)"] \\ GB \arrow[r, "\psi_B"'] & FB \end{tikzcd}$ $G(R)\varphi_B \sqsubseteq \varphi_A F(R)$ and $G(R)\psi_B \sqsubseteq \psi_A F(R)$ give $G(R)(\varphi_B \cup \psi_B) \sqsubseteq (\varphi_A \cup \psi_A)F(R)$	
4	a relator K $K(\varphi)$	$\begin{tikzcd} KGA \arrow[r, "K(G(R))"] \arrow[d, "K(\varphi_A)"'] \arrow[dr, phantom, "\sqsubseteq"] & KGB \arrow[d, "K(\varphi_B)"] \\ KFA \arrow[r, "K(F(R))"'] & KFB \end{tikzcd}$ $G(R)\varphi_B \sqsubseteq \varphi_A F(R)$ gives $K(G(R))K(\varphi_B) \sqsubseteq K(\varphi_A)K(F(R))$	
5	product $\varphi\times\psi$	$\begin{tikzcd} GA\times G'A \arrow[r, "\varphi_A\times\psi_A"] \arrow[d, "G(R)\times G'(R)"'] \arrow[dr, phantom, "\sqsubseteq"] & FA\times F'A \arrow[d, "F(R)\times F'(R)"] \\ GB\times G'B \arrow[r, "\varphi_B\times\psi_B"'] & FB\times F'B \end{tikzcd}$ $\varphi : G \Rightarrow F$ and $\psi : G' \Rightarrow F'$ give $\varphi\times\psi : G\times G' \Rightarrow F\times F'$ $(R\times S)(U\times V) = (RU)\times(SV)$ and monotonicity in both slots — the row above's K applied to the inequation, run on a bifunctor; the fork is the derived case $\langle\varphi,\psi\rangle = \triangleleft(\varphi\times\psi)$, and its ONE cost is the lax copy law $R\triangleleft \sqsubseteq \triangleleft(R\times R)$ — $R \triangleleft \leq \triangleleft (R \otimes R)$, $R \multimap \leq \multimap$	
6	coproduct $\varphi+\psi$	$\begin{tikzcd} GA+G'A \arrow[r, "\varphi_A+\psi_A"] \arrow[d, "G(R)+G'(R)"'] \arrow[dr, phantom, "\sqsubseteq"] & FA+F'A \arrow[d, "F(R)+F'(R)"] \\ GB+G'B \arrow[r, "\varphi_B+\psi_B"'] & FB+F'B \end{tikzcd}$ $\varphi : G \Rightarrow F$ and $\psi : G' \Rightarrow F'$ give $\varphi+\psi : G+G' \Rightarrow F+F'$ $(R+S)(U+V) = (RU)+(SV)$ and monotonicity in both slots; the co-fork is the derived case $[\varphi,\psi] = (\varphi+\psi)\triangledown$, and $\triangledown$ costs nothing	
7	meet — **fails** $\varphi\cap\psi$	the step it would need is $(\varphi_A F(R)) \cap (\psi_A F(R)) \sqsubseteq (\varphi\cap\psi)_A F(R)$, the wrong direction of $R\ (S \cap T) \sqsubseteq (R\ S) \cap (R\ T)$ a counterexample: $aa = (0,1) \wedge bb = (0,0) \wedge (R \times R)(aa,bb) \wedge b_1 = \{0\} \wedge b_1 = E(\pi_1 \cap \pi_2)\{bb\} \wedge a_1 = \varnothing \wedge a_1 = E(\pi_1 \cap \pi_2)\{aa\} \wedge b_2 = \varnothing \wedge b_2 = E(R)(a_1)$ $\Longrightarrow \neg((R \times R)\ (\pi_1 \cap \pi_2) \sqsubseteq (\pi_1 \cap \pi_2)\ R)$	

8 Each row is the general slide $R_A X \sqsubseteq X' R_B$ and $R_B Y \sqsubseteq Y' R_C$ give $R_A(XY) \sqsubseteq (X'Y')R_C$ at $R_A,R_B := G(R),F(R)$ and $X,X' := \varphi_B,\varphi_A$, quantified over R; at $X'=X$ with R_A,R_B endorelations it is the monotonicity reading instead — the $\forall R$ sits outside the law, and is the only difference

§5.7.1. meet is not closed in LaT

(5.7.1a)

counterexample — φ,ψ : G⇒F lax natural does NOT give φ∩ψ lax natural

$$\begin{tikzcd}
A\times A \arrow[r, "\pi_1\cap\pi_2"] \arrow[d, "R\times R"'] \arrow[dr, phantom, "\not\sqsubseteq"] & A \arrow[d, "R"] \\
B\times B \arrow[r, "\pi_1\cap\pi_2"'] & B
\end{tikzcd}$$

$(0,1)$, $\varnothing$, $(0,0)$, $\{0\}$, $\varnothing$

A=B≜{0,1}, R≜{(0,0),(1,0)}, φ≜π₁∩π₂ : Δ⇒Id

§5.7.2. Two stacked towers

(5.7.2a)

		lower — inside ONE allegory	upper — between allegories
1	0-cell	an object A	an allegory $\mathcal{C}$ — **a whole lower tower**
2	1-cell	a relation R : A→B	a relator F : $\mathcal{C}\to\mathcal{D}$
3	2-cell	R⊑S	a LaT φ : G⇒F ≜ lax
4	order	R⊑S	φ⊑ψ componentwise
5	union	R ∪ S	φ ∪ ψ, **survives** lax(φ) ∧ lax(ψ) ⟹ G(R) (φ_B ∪ ψ_B) ⊑ (φ_A ∪ ψ_A) F(R)
6	product	R×S ≜ ×	φ×ψ, **survives**; a TENSOR, not a categorical product; SYMMETRIC ⟨R,S⟩ π₁ = dom(S) R , ⟨R,S⟩ π₂ = dom(R) S lax(φ) ∧ lax(ψ) ⟹ (G(R) × G'(R)) (φ_B × ψ_B) ⊑ (φ_A × ψ_A) (F(R) × F'(R))
7	coproduct	R+S	φ+ψ, **survives**; a BIPRODUCT lax(φ) ∧ lax(ψ) ⟹ (G(R) + G'(R)) (φ_B + ψ_B) ⊑ (φ_A + ψ_A) (F(R) + F'(R))
8	meet	R∩S	φ∩ψ componentwise, **fails** aa = (0,1) ∧ bb = (0,0) ∧ (R × R)(aa,bb) ∧ b₁ = {0} ∧ b₁ = E(π₁ ∩ π₂){bb} ∧ a₁ = ∅ ∧ a₁ = E(π₁ ∩ π₂){aa} ∧ b₂ = ∅ ∧ b₂ = E(R)(a₁) ⟹ ¬((R × R) (π₁ ∩ π₂) ⊑ (π₁ ∩ π₂) R)
9	converse	R°	φ°, **fails**, oplax: φ_A°G(R)⊑F(R)φ_B°
10	zero object	z with 𝟙 z=𝟘	the constant relator at such a z, initial and terminal

§5.7.3. R⊑S is a special case of LaT

(5.7.3a)

$$\begin{tikzcd}
GA \arrow[r, "\varphi_A"] \arrow[d, "G(R)"'] \arrow[dr, phantom, "\sqsubseteq"] & FA \arrow[d, "F(R)"] \\
GB \arrow[r, "\varphi_B"'] & FB
\end{tikzcd}
\Longrightarrow
\begin{tikzcd}
A \arrow[r, "\varphi_X"] \arrow[d, "\mathbb{1}"'] \arrow[dr, phantom, "\sqsubseteq"] & B \arrow[d, "\mathbb{1}"] \\
A \arrow[r, "\varphi_Y"'] & B
\end{tikzcd}
\Longrightarrow
\begin{tikzcd}
A \arrow[r, bend left, "\varphi_Y"] \arrow[r, bend right, "\varphi_X"'] \arrow[r, phantom, "\sqsubseteq"] & B
\end{tikzcd}$$

G≜const A, F≜const B — the relators X↦A and X↦B, each sending every arrow to 𝟙 ≜ lax

with $\mathcal{C}$ two objects and one non-identity arrow X→Y, a LaT const A⇒const B is the pair φ_X,φ_Y : A→B with φ_Y⊑φ_X — the datum R⊑S

not every LaT is constant: ∋ : P⇒Id X ⊑ P(R) ⟺ X ∋ ⊑ ∋ R ∧ X° ∋ ⊑ ∋ R° π₂ : (A×−)⇒Id (𝟙 × ∋) π₂ = π₂ ∋

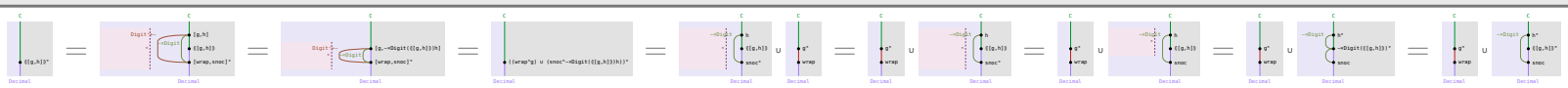
◁ : Id⇒Δ R ◁ ≤ ◁ (R ⊗ R) , R ⊸ ≤ ⊸

§6. Recursive Programs

§6.1. Digits of a number

⦇[g,h]⦈° = (g° wrap) ∪ (h° −×Digit(⦇[g,h]⦈°) snoc) (6.1a)

the converse of a fold over decimals undoes the algebra **[g,h]** and then either **wrap**, or the fold's converse on the front and **snoc**



(1) (2) (3) (4) (5) (6) (7) (8) (9)

1	⦇[g,h]⦈°	⦇φ⦈ = [wrap,snoc]° Digit⁺+−(−×Digit(⦇φ⦈)) φ
2	= ([wrap,snoc]° Digit⁺+−(−×Digit(⦇[g,h]⦈)) [g,h])°	(R + S) [P,Q] = [R P,S Q]
3	= ([wrap,snoc]° [g,−×Digit(⦇[g,h]⦈) h])°	[U,V]° [R,S] = (U° R) ∪ (V° S)
4	= ((wrap° g) ∪ (snoc° −×Digit(⦇[g,h]⦈) h))°	(R ∪ S)° = S° ∪ R°
5	= (snoc° −×Digit(⦇[g,h]⦈) h)° ∪ (wrap° g)°	R ∪ S = S ∪ R
6	= (wrap° g)° ∪ (snoc° −×Digit(⦇[g,h]⦈) h)°	(R S)° = S° R°
7	= (g° wrap) ∪ ((−×Digit(⦇[g,h]⦈) h)° snoc)	(R S)° = S° R°
8	= (g° wrap) ∪ (h° −×Digit(⦇[g,h]⦈)° snoc)	−×Digit(R)° = −×Digit(R°)
9	= (g° wrap) ∪ (h° −×Digit(⦇[g,h]⦈°) snoc)	

val° = (embed° wrap) ∪ (op° −×Digit(val°) snoc)

a number is read back into digits either as one nonzero digit, or by splitting off its last digit and reading back the rest

(6.1b)

1 op°(m,p) ⟺ π₁(p) = m/10 ∧ π₂(p) = m % 10

op° splits a number into its quotient and remainder by 10

2 (∃p. 0 < π₁(p) ∧ op°(m,p)) ⟺ 10 ≤ m

op° gives a pair with a nonzero first component exactly at the numbers with two or more digits

3 (∃d. embed°(m,d)) ⟺ 0 < m ∧ m < 10

embed° gives a digit exactly at the one-digit numbers

§6.2. Least fixed points

(6.2a)

`Monotonic(φ) ≜ ∀X. ∀Y. X ⊑ Y ⟹ φ(X) ⊑ φ(Y)` — φ maps the hom-set `A→B` into itself with `X⊑Y⟹φ(X)⊑φ(Y)`.

`(μX : φ(X))` is the least `X : A→B` with `φ(X)⊑X`.

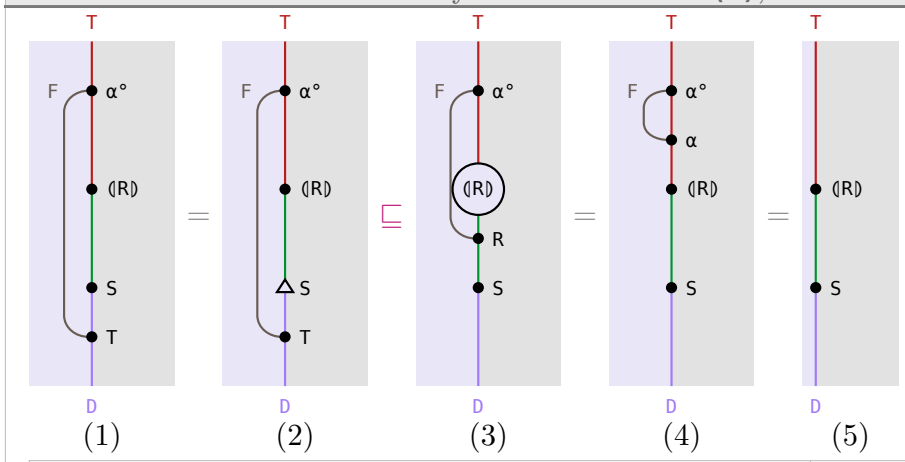
(6.2b)

	the law	what it says
1	`φ(Y) ⊑ Y ⟹ (μX : φ(X)) ⊑ Y`	to bound `(μX : φ(X))` above, exhibit one `Y` the body does not grow past — the half §6.3 and every chapter after it uses
2	`Monotonic(φ) ⟹ φ(μX : φ(X)) = (μX : φ(X))`	**Knaster–Tarski**: the least solution of `φ(X)⊑X` already solves `φ(X)=X`, so the least prefix point and the least fixed point are one relation
3	`α° F(X) R ⊑ X ⟹ ⦇R⦈ ⊑ X` (6.2)	a relation `X` that contains one unfolding of the fold's recursion at `X` contains the fold
4	`X ⊑ α° F(X) R ⟹ X ⊑ ⦇R⦈` (6.3)	a relation `X` contained in one unfolding of the fold's recursion at `X` is contained in the fold

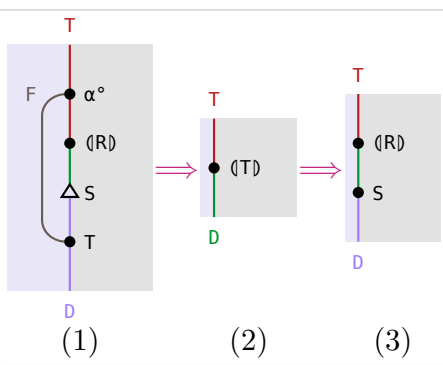
(6.2c)

`F(S) T ⊑ R S ⟹ ⦇T⦈ ⊑ ⦇R⦈ S`

if **S** followed by **R** absorbs **T** after **F(S)**, the fold of **T** is below the fold of **R** followed by **S**



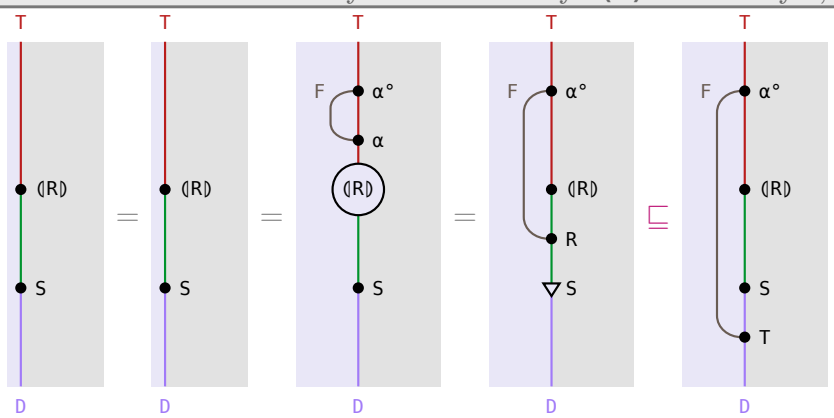
1	`α° F(⦇R⦈ S) T`	
2	`= α° F(⦇R⦈) F(S) T`	`F(S) T ⊑ R S`
3	`⊑ α° F(⦇R⦈) R S`	`α ⦇R⦈ = F(⦇R⦈) R`
4	`= α° α ⦇R⦈ S`	`α° α = 𝟙`
5	`= ⦇R⦈ S`	



1	`F(S) T ⊑ R S`	`F(S) T ⊑ R S` `⟹ α° F(⦇R⦈ S) T ⊑ ⦇R⦈ S`
2	`⟹ α° F(⦇R⦈ S) T ⊑ ⦇R⦈ S`	`α° F(X) R ⊑ X` `⟹ ⦇R⦈ ⊑ X`
3	`⟹ ⦇T⦈ ⊑ ⦇R⦈ S`	

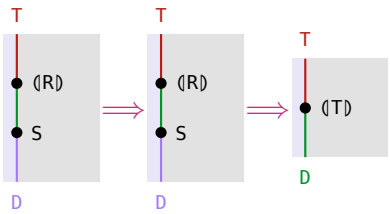
$$\mathbf{R}\ \mathbf{S} \sqsubseteq \mathbf{F}(\mathbf{S})\ \mathbf{T} \implies (\!|\mathbf{R}|\!)\ \mathbf{S} \sqsubseteq (\!|\mathbf{T}|\!) \tag{6.2d}$$

if **R** followed by **S** is absorbed by **F(S)** followed by **T**, the fold of **R** followed by **S** is below the fold of **T**



(1) (2) (3) (4) (5)

1	$(\!\|R\|\!)\ S$	
2	$= (\!\|R\|\!)\ S$	$\alpha^\circ\ \alpha = \mathbb{1}$
3	$= \alpha^\circ\ \alpha\ (\!\|R\|\!)\ S$	$\alpha\ (\!\|R\|\!) = F((\!\|R\|\!))\ R$
4	$= \alpha^\circ\ F((\!\|R\|\!))\ R\ S$	$R\ S \sqsubseteq F(S)\ T$
5	$\sqsubseteq \alpha^\circ\ F((\!\|R\|\!)\ S)\ T$	



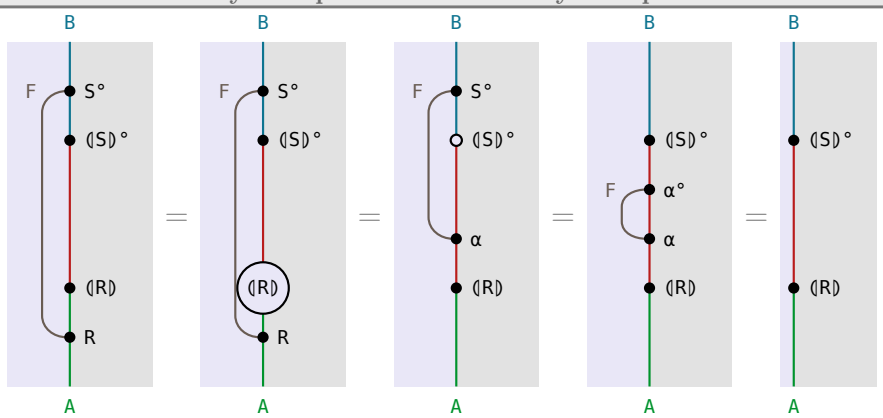
(1) (2) (3)

1	$R\ S \sqsubseteq F(S)\ T$	$R\ S \sqsubseteq F(S)\ T$ $\implies (\!\|R\|\!)\ S \sqsubseteq \alpha^\circ\ F((\!\|R\|\!)\ S)\ T$
2	$\implies (\!\|R\|\!)\ S \sqsubseteq \alpha^\circ\ F((\!\|R\|\!)\ S)\ T$	$X \sqsubseteq \alpha^\circ\ F(X)\ R$ $\implies X \sqsubseteq (\!\|R\|\!)$
3	$\implies (\!\|R\|\!)\ S \sqsubseteq (\!\|T\|\!)$	

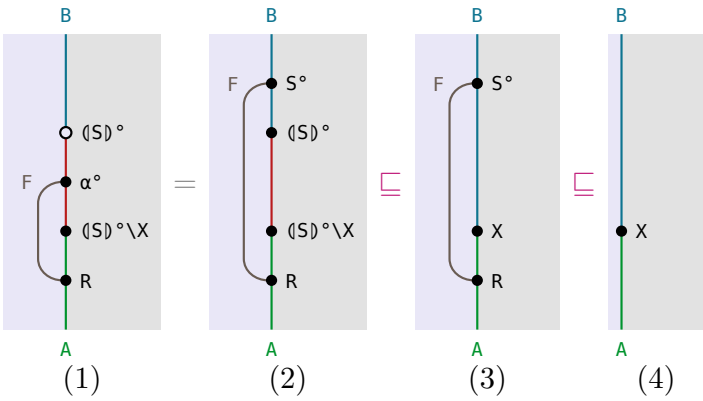
§6.3. Hylomorphisms

⦇S⦈° ⦇R⦈ = (μX : S° F(X) R) (6.3a)

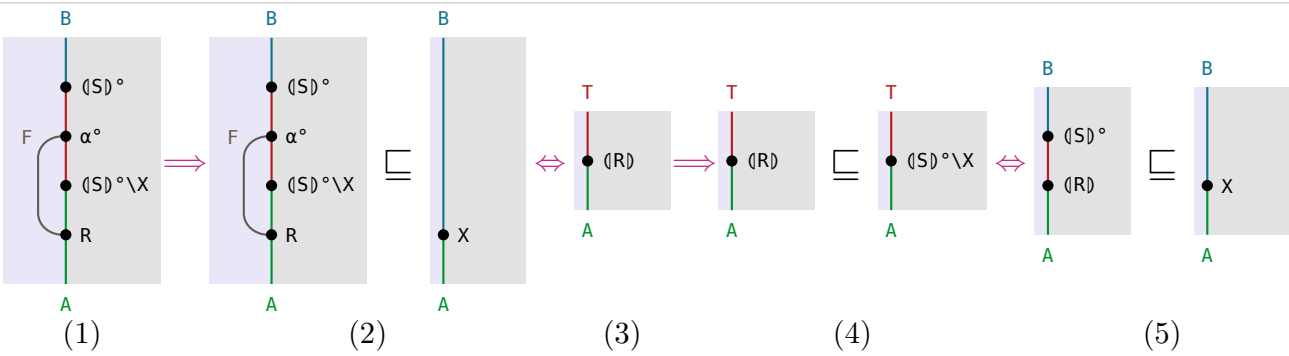
hylomorphism theorem: a hylomorphism is the least fixed point of a certain recursion equation



1	S° F(⦇S⦈° ⦇R⦈) R	
2	= S° F(⦇S⦈°) F(⦇R⦈) R	α ⦇R⦈ = F(⦇R⦈) R
3	= S° F(⦇S⦈°) α ⦇R⦈	⦇R⦈° α° = R° F(⦇R⦈°)
4	= ⦇S⦈° α° α ⦇R⦈	α° α = 𝟙
5	= ⦇S⦈° ⦇R⦈	



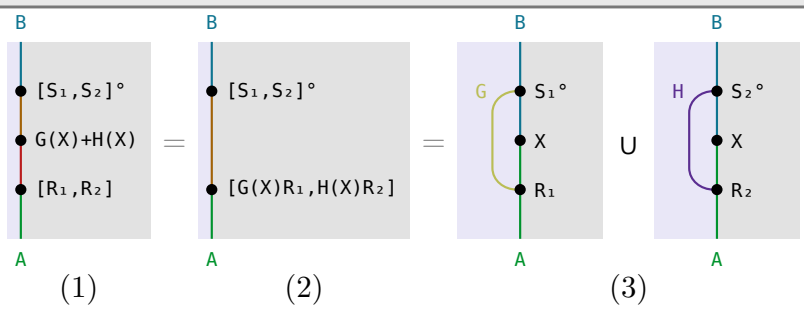
1	⦇S⦈° α° F(⦇S⦈° \ X) R	⦇R⦈° α° = R° F(⦇R⦈°)
2	= S° F(⦇S⦈° (⦇S⦈° \ X)) R	S (S \ R) ⊑ R
3	⊑ S° F(X) R	S° F(X) R ⊑ X
4	⊑ X	



1	S° F(X) R ⊑ X	S° F(X) R ⊑ X ⟹ ⦇S⦈° α° F(⦇S⦈° \ X) R ⊑ X
2	⟹ ⦇S⦈° α° F(⦇S⦈° \ X) R ⊑ X	X ⊑ S \ R ⟺ S X ⊑ R
3	⇔ α° F(⦇S⦈° \ X) R ⊑ ⦇S⦈° \ X	α° F(X) R ⊑ X ⟹ ⦇R⦈ ⊑ X
4	⟹ ⦇R⦈ ⊑ ⦇S⦈° \ X	X ⊑ S \ R ⟺ S X ⊑ R
5	⇔ ⦇S⦈° ⦇R⦈ ⊑ X	

$$(\![S_1,S_2]\!)^\circ\ (\![R_1,R_2]\!) = (\mu X : (S_1{}^\circ\ G(X)\ R_1) \cup (S_2{}^\circ\ H(X)\ R_2)) \tag{6.3b}$$

when both algebras are case splits over the same sum, the recursion runs each case on its own and unites the results



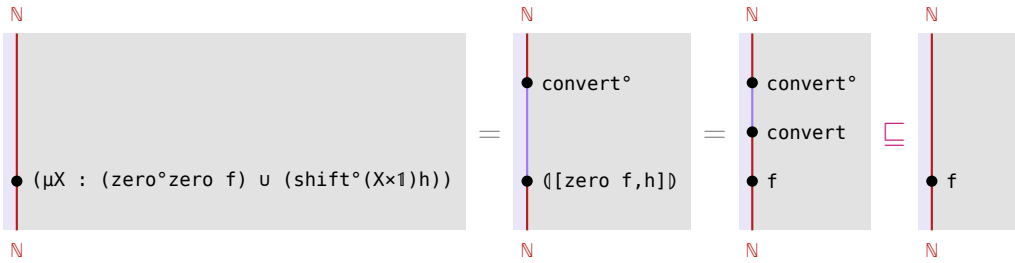
(1) (2) (3)

1	$[S_1,S_2]^\circ\ (G(X) + H(X))\ [R_1,R_2]$	$(R + S)\ [P,Q] = [R\ P, S\ Q]$
2	$= [S_1,S_2]^\circ\ [G(X)\ R_1, H(X)\ R_2]$	$[U,V]^\circ\ [R,S] = (U^\circ\ R) \cup (V^\circ\ S)$
3	$= (S_1{}^\circ\ G(X)\ R_1) \cup (S_2{}^\circ\ H(X)\ R_2)$	

§6.4. Fast exponentiation and modulus computation

shift f = (f × 𝟙) h ⟹ (μX : (zero° zero f) ∪ (shift° (X × 𝟙) h)) ⊑ f (6.4a)

once **f** turns **shift** into **h**, the recursion that halves the argument at each step computes **f**



1	(μX : (zero° zero f) ∪ (shift° (X × 𝟙) h))	convert° ⦇[g,h]⦈ = (μX : (zero° g) ∪ (shift° (X × 𝟙) h))
2	= convert° ⦇[zero f,h]⦈	zero f = g ∧ shift f = (f × 𝟙) h ⟹ convert f = ⦇[g,h]⦈
3	= convert° convert f	Simple(convert)
4	⊑ f	

(μX : (zero° one) ∪ (shift° (X × 𝟙) op(a))) ⊑ exp(a) (6.4b)

the recursion that halves the exponent at each step computes **a** to the power **b**

zero exp(a) = one

a to the power 0 is 1

shift exp(a) = (exp(a) × 𝟙) op(a)

a to the power 2n+d is op(a) of a to the power n and d

(μX : (zero° zero) ∪ (shift° (X × 𝟙) op(b))) ⊑ mod(b) (6.4c)

the recursion that halves **a** at each step computes the remainder of **a** by **b**

zero mod(b) = zero

0 mod b is 0

shift mod(b) = (mod(b) × 𝟙) op(b)

(2a+d) mod b is op(b) of a mod b and d

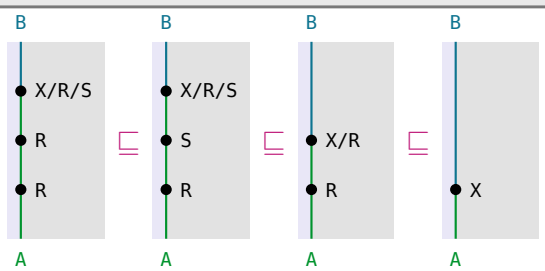
§6.5. Unique fixed points

$$\mathsf{R\ R \sqsubseteq S\ R \implies (X\ /\ R\ /\ S)\ R\ R \sqsubseteq X} \tag{6.5a}$$

((X/R)/S) followed by RR stays below X when RR⊑SR

$$\mathsf{inductive(S) \land R\ R \sqsubseteq S\ R \implies inductive(R)}$$

R is inductive when RR⊑SR for an inductive S



(1) (2) (3) (4)

1	(X / R / S) R R	R R ⊑ S R
2	⊑ (X / R / S) S R	(R / S) S ⊑ R
3	⊑ (X / R) R	(R / S) S ⊑ R
4	⊑ X	

§6.5.1. Membership

$$\mathsf{member = \emptyset} \tag{6.5.1a}$$

a constant relator records no elements, so its membership is empty

$$\mathsf{member = [member,member]}$$

a member of F+G is a member of whichever summand the value is in

$$\mathsf{member = (\pi_1\ member) \cup (\pi_2\ member)}$$

a member of F×G is a member of either component

§6.5.2. Consequences

$$\mathsf{inductive(S\ member) \land X = S\ F(X)\ R \land Y = S\ F(Y)\ R \implies X = Y} \tag{6.5.2a}$$

when S member is inductive, two solutions X, Y of X=SF(X)R are equal

$$\mathsf{inductive(S\ member) \land Entire(S) \land Entire(R) \land S\ F(X)\ R \sqsubseteq X \implies Entire(X)} \tag{6.5.2b}$$

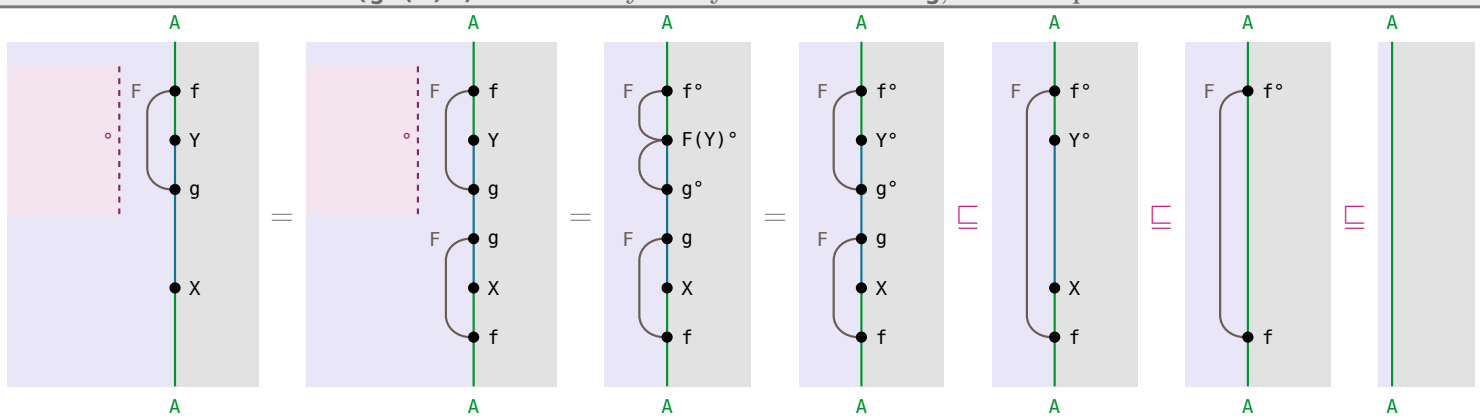
when S member is inductive and S, R are entire, a pre-fixed point of X↦SF(X)R is entire

$$\mathsf{inductive(g\ member) \land Map(g) \land Map(f) \land X = g\ F(X)\ f \implies Map(X)} \tag{6.5.2c}$$

when g member is inductive and f, g are maps, the solution of X=gF(X)f is a map

$$\mathsf{X = g\ F(X)\ f \land Simple(g) \land Simple(f) \land Y^\circ\ X \sqsubseteq \mathbb{1} \implies (g\ F(Y)\ f)^\circ\ X \sqsubseteq \mathbb{1}}$$

(gF(Y)f)° followed by X stays below 𝟙 when g, f are simple and Y°X⊑𝟙

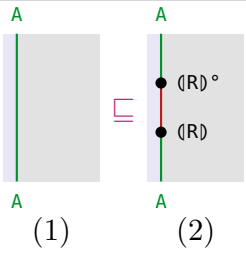


(1) (2) (3) (4) (5) (6) (7)

1	(g F(Y) f)° X	X = g F(X) f
2	= (g F(Y) f)° g F(X) f	(R S)° = S° R°
3	= f° F(Y)° g° g F(X) f	PreservesRecip(F)
4	= f° F(Y°) g° g F(X) f	Simple(g)
5	⊑ f° F(Y° X) f	Y° X ⊑ 𝟙
6	⊑ f° f	Simple(f)
7	⊑ 𝟙	

inductive(R° member) ∧ 𝟙 ⊑ R° R ⟹ 𝟙 ⊑ ⦇R⦈° ⦇R⦈ (6.5.2d)

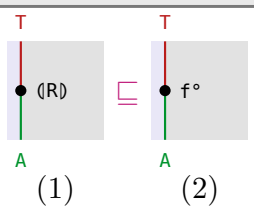
when **R°member** is inductive and **R** is surjective, the fold **⦇R⦈** is surjective



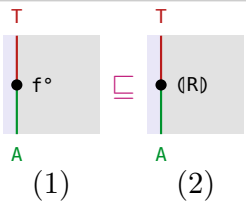
1	𝟙	inductive(S member) ∧ Entire(S) ∧ Entire(R) ∧ S F(X) R ⊑ X ⟹ Entire(X) at S=R°
2	⊑ ⦇R⦈° ⦇R⦈	

Map(f) ∧ R f ⊑ F(f) α ∧ 𝟙 ⊑ R° R ⟹ f° = ⦇R⦈ (6.5.2e)

a map **f** with **Rf⊑F(f)α**, for a surjective **R**, has the fold **⦇R⦈** as its converse



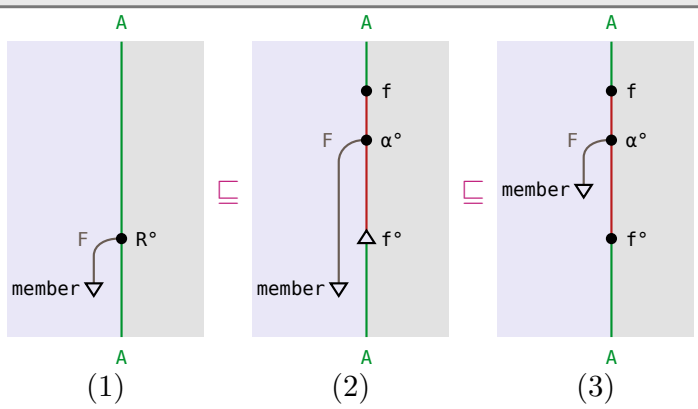
1	⦇R⦈	shunting f, fusion, Rf⊑F(f)α
2	⊑ f°	



1	f°	inductive(R° member) ∧ 𝟙 ⊑ R° R ⟹ 𝟙 ⊑ ⦇R⦈° ⦇R⦈ and ⦇R⦈f⊑𝟙
2	⊑ ⦇R⦈	

Map(f) ∧ R f ⊑ F(f) α ⟹ R° member ⊑ f α° member f° (6.5.2f)

R°member is below **α°member** conjugated by **f**, so it is inductive when **α°member** is



1	R° member	Map(f) ∧ R f ⊑ F(f) α ⟹ R° ⊑ f α° F(f°)
2	⊑ f α° F(f°) member	F(R) member ⊑ member R
3	⊑ f α° member f°	

§6.6. Sorting by selection

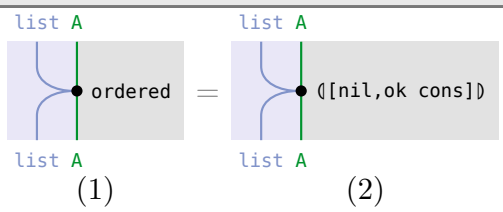
§6.6.1. The function sort

(6.6.1a)

	definition	type	meaning
1	`ordered[] = true, ordered([a]+xs) = (∀x. x ∈ xs ⟹ R(a,x)) ∧ ordered(xs)`	`[A]→[A]`	every element is R-below each element after it
2	`(a,xs) ok q ⟺ (a,xs) = q ∧ (∀x. x ∈ xs ⟹ R(a,x))`	`A×[A]→A×[A]`	the coreflexive passing `(a,x)` when `a` is R-below every element of `x`

`ordered = ⦇[nil,ok cons]⦈` (6.6.1b)

a list is sorted exactly when the fold that rebuilds it passes **ok** at every **cons**, i.e. each head is **R**-below every element after it

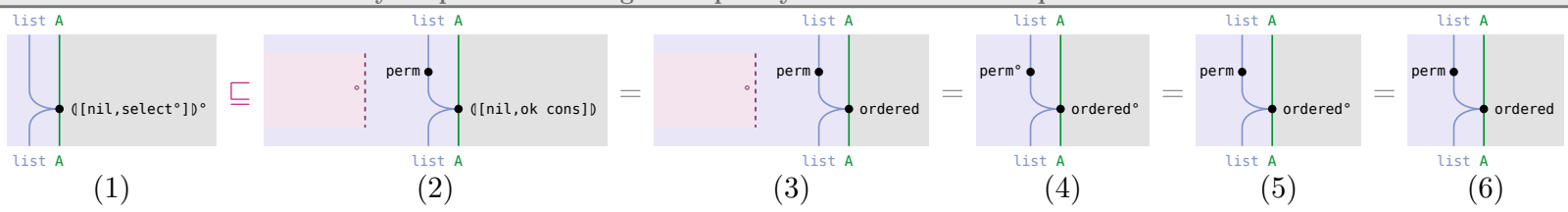


1	`ordered`	fold uniqueness
2	`= ⦇[nil,ok cons]⦈`	

§6.6.2. Selection sort

`select° ⊑ ok cons perm ⟹ ⦇[nil,select°]⦈° ⊑ perm ordered` (6.6.2a)

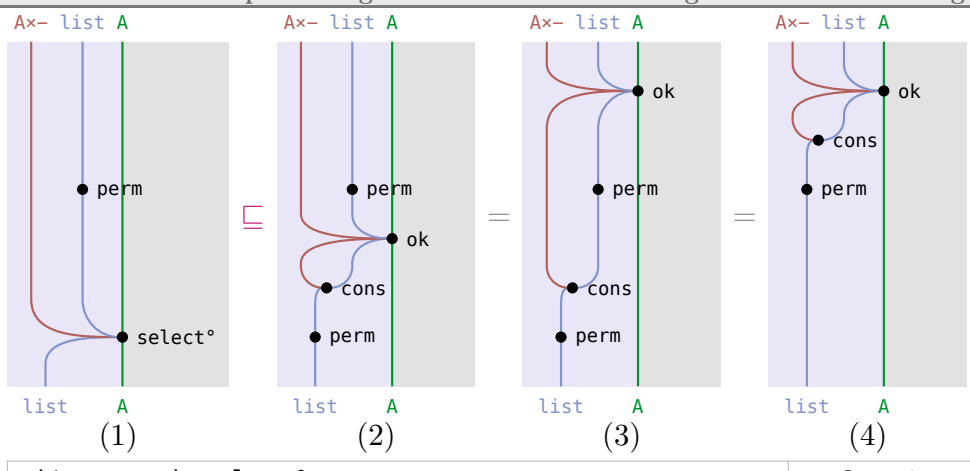
every output of unfolding the input by **select** is a sorted permutation of it



1	`⦇[nil,select°]⦈°`	`select° ⊑ ok cons perm` `⟹ ⦇[nil,select°]⦈° ⊑ (⦇[nil,ok cons]⦈ perm)°`
2	`⊑ (⦇[nil,ok cons]⦈ perm)°`	(6.6.1b) `ordered = ⦇[nil,ok cons]⦈`
3	`= (ordered perm)°`	`(R S)° = S° R°`
4	`= perm° ordered°`	`perm° = perm`
5	`= perm ordered°`	`ordered° = ordered`
6	`= perm ordered`	

`select° ⊑ ok cons perm ⟹ (𝟙 × perm) select° ⊑ ok cons perm` (6.6.2b)

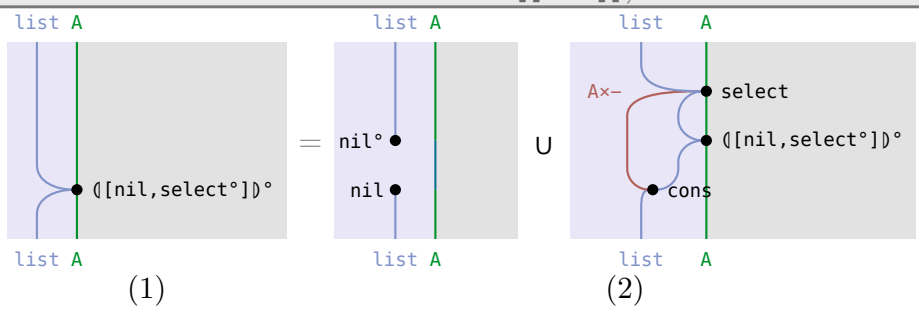
permuting the tail and then undoing **select** lands among the **ok** conses of a permutation



1	`(𝟙 × perm) select°`	`select° ⊑ ok cons perm`
2	`⊑ (𝟙 × perm) ok cons perm`	`ok (𝟙 × perm) = (𝟙 × perm) ok`
3	`= ok (𝟙 × perm) cons perm`	`cons perm = (𝟙 × perm) cons perm`
4	`= ok cons perm`	

⦇[nil,select°]⦈° = (nil° nil) ∪ (select (𝟙 × ⦇[nil,select°]⦈°) cons) (6.6.2c)

selection sort returns **[]** on **[]**, and otherwise selects **(a,y)**, sorts **y** and conses **a** back on

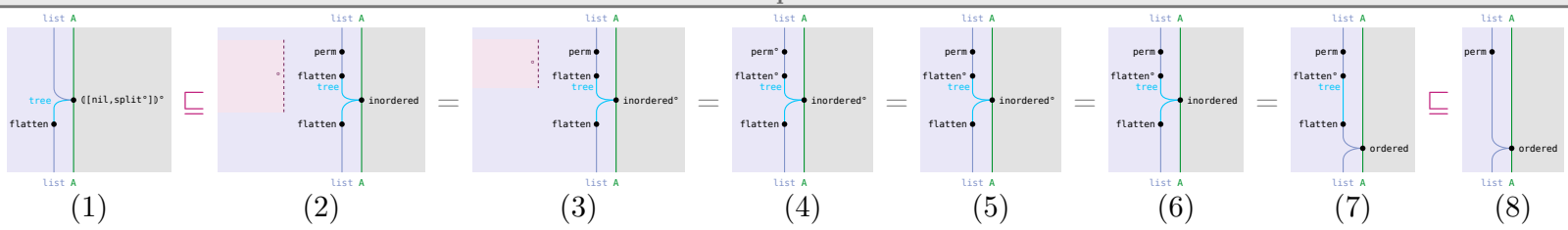


1	⦇[nil,select°]⦈°	unfolding the converse of a fold
2	= (nil° nil) ∪ (select (𝟙 × ⦇[nil,select°]⦈°) cons)	

§6.6.3. Quicksort

(∀a. ∀b. ∀c. R(a,b) ⟹ R(b,c) ⟹ R(a,c)) ∧ split° ⊑ check' join perm ⟹ ⦇[nil,split°]⦈° flatten ⊑ perm ordered (6.6.3a)

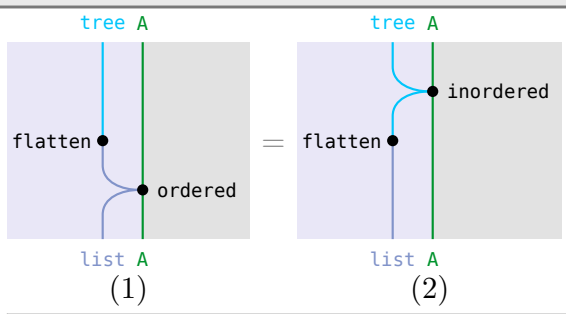
every output of unfolding the input by **split** into a tree and flattening that tree is a sorted permutation of the input



1	⦇[nil,split°]⦈° flatten	split° ⊑ check' join perm ⟹ ⦇[nil,split°]⦈° flatten ⊑ (inordered flatten perm)° flatten
2	⊑ (inordered flatten perm)° flatten	(R S)° = S° R°
3	= (flatten perm)° inordered° flatten	(R S)° = S° R°
4	= perm° flatten° inordered° flatten	perm° = perm
5	= perm flatten° inordered° flatten	inordered° = inordered
6	= perm flatten° inordered flatten	(6.6.3b) (∀a. ∀b. ∀c. R(a,b) ⟹ R(b,c) ⟹ R(a,c)) ⟹ flatten ordered = inordered flatten
7	= perm flatten° flatten ordered	flatten° flatten ⊑ 𝟙
8	⊑ perm ordered	

(∀a. ∀b. ∀c. R(a,b) ⟹ R(b,c) ⟹ R(a,c)) ⟹ flatten ordered = inordered flatten (6.6.3b)

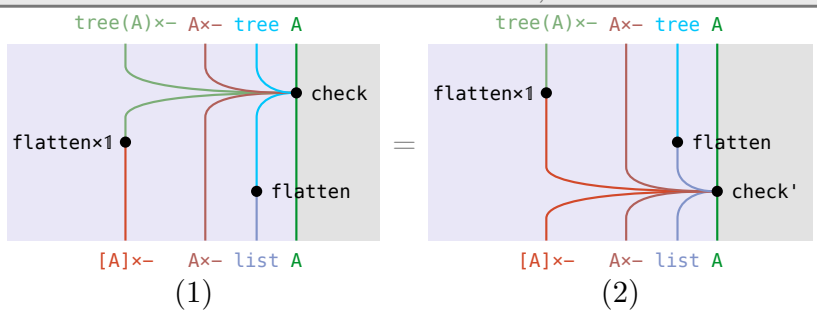
flattening a tree and then testing the list for order is the same as testing the tree with **inordered** and then flattening



1	flatten ordered	R transitive: a flattened tree is ordered iff every node passes check
2	= inordered flatten	

check (flatten × (𝟙 × flatten)) = (flatten × (𝟙 × flatten)) check' (6.6.3c)

testing a fork with **check** and then flattening both subtrees equals flattening both subtrees and then testing with **check'**, since **intree** of a tree is **inlist** of its flattening

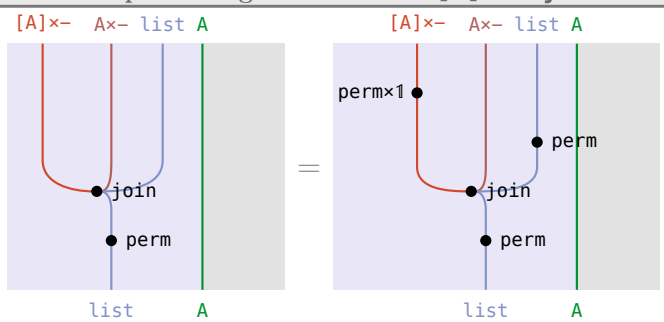


(1) (2)

1	check (flatten × (𝟙 × flatten))	b intree x ⟺ b inlist flatten(x)
2	= (flatten × (𝟙 × flatten)) check'	

join perm = (perm × (𝟙 × perm)) join perm (6.6.3d)

permuting the list **x ++ [a] ++ y** is the same as first permuting **x** and **y** and then permuting the joined list

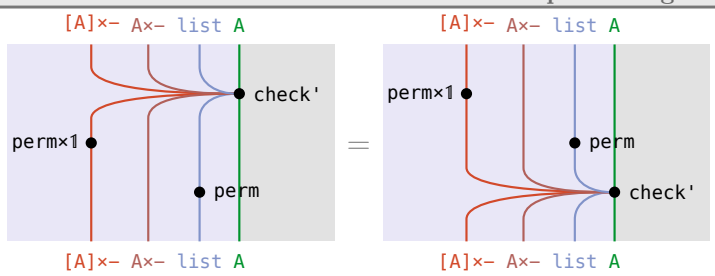


(1) (2)

1	join perm	perm is transitive and closed under ++
2	= (perm × (𝟙 × perm)) join perm	

check' (perm × (𝟙 × perm)) = (perm × (𝟙 × perm)) check' (6.6.3e)

testing a fork of lists with **check'** and then permuting both parts equals permuting both parts and then testing, since permuting a list leaves its members unchanged

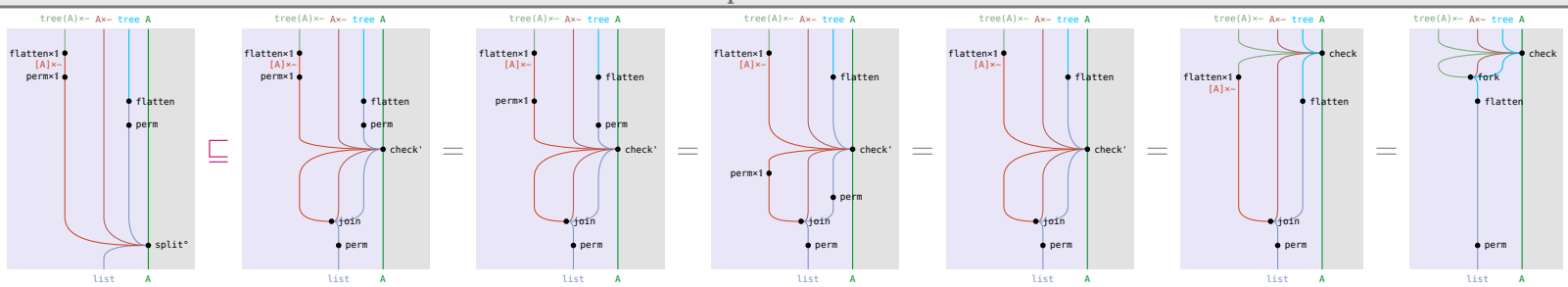


(1) (2)

1	check' (perm × (𝟙 × perm))	b inlist x ⟺ b inlist perm(x)
2	= (perm × (𝟙 × perm)) check'	

split° ⊑ check' join perm ⟹ ((flatten perm) × (𝟙 × (flatten perm))) split° ⊑ check fork flatten perm (6.6.3f)

undoing **split** and then flattening and permuting both parts lands among the **check**ed forks whose flattening is permuted

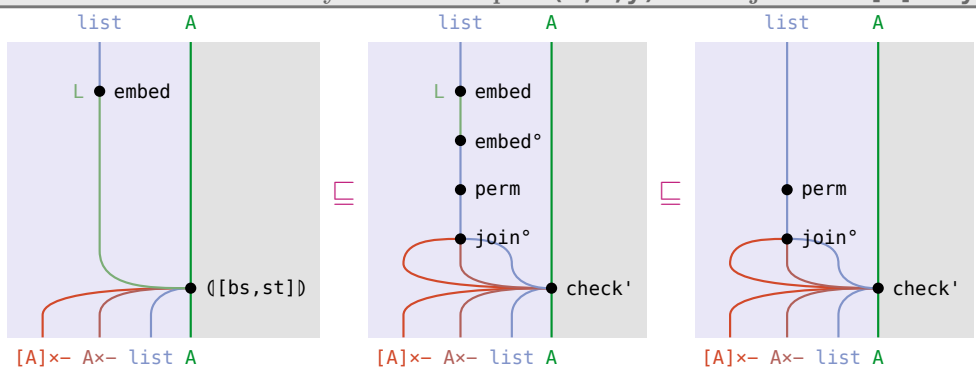


(1) (2) (3) (4) (5) (6) (7)

1	((flatten perm) × (𝟙 × (flatten perm))) split°	split° ⊑ check' join perm
2	⊑ ((flatten perm) × (𝟙 × (flatten perm))) check' join perm	
3	= (flatten × (𝟙 × flatten)) (perm × (𝟙 × perm)) check' join perm	(6.6.3e) check' (perm × (𝟙 × perm)) = (perm × (𝟙 × perm)) check'
4	= (flatten × (𝟙 × flatten)) check' (perm × (𝟙 × perm)) join perm	(6.6.3d) join perm = (perm × (𝟙 × perm)) join perm
5	= (flatten × (𝟙 × flatten)) check' join perm	(6.6.3c) check (flatten × (𝟙 × flatten)) = (flatten × (𝟙 × flatten)) check'
6	= check (flatten × (𝟙 × flatten)) join perm	fork flatten = (flatten × (𝟙 × flatten)) join
7	= check fork flatten perm	

bs ⊑ wrap perm join° check' ∧ (𝟙 × (perm join° check')) st ⊑ cons perm join° check' ⟹ embed ⦇[bs,st]⦈ ⊑ perm join° check' (6.6.3g)

turning a list into a non-empty list and folding it with any **[bs, st]** meeting the two conditions below gives only **check**ed triples **(x,a,y)** whose join **x ⧺ [a] ⧺ y** is a permutation of the list

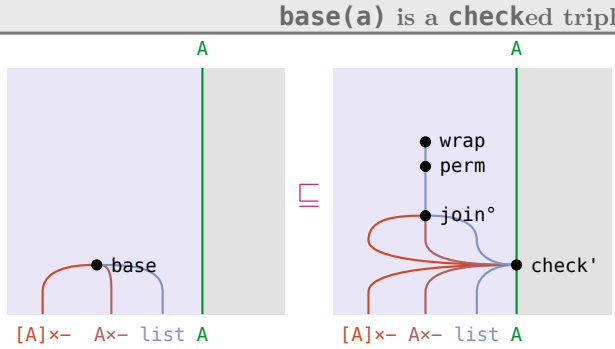


(1) (2) (3)

1	embed ⦇[bs,st]⦈	bs ⊑ wrap perm join° check' ∧ (𝟙 × (perm join° check')) st ⊑ cons perm join° check' ⟹ embed ⦇[bs,st]⦈ ⊑ embed embed° perm join° check'
2	⊑ embed embed° perm join° check'	embed embed° perm join° check' ⊑ perm join° check'
3	⊑ perm join° check'	

base ⊑ wrap perm join° check' (6.6.3h)

base(a) is a **check**ed triple whose join is a permutation of the one-element list **[a]**

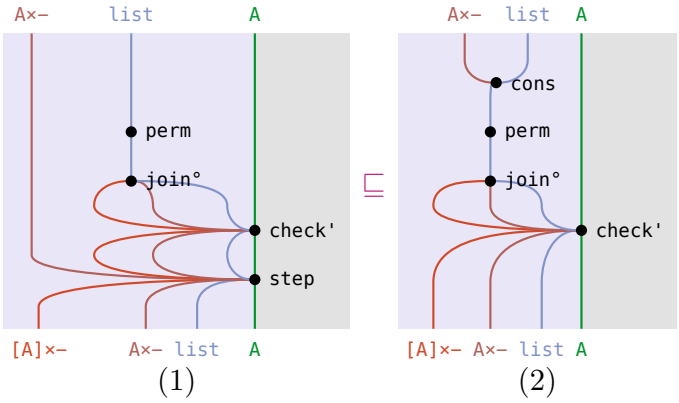


(1) (2)

1	base	[] ⧺ [a] ⧺ [] = [a], and check' holds on empty lists
2	⊑ wrap perm join° check'	

connected(R) ⟹ (𝟙 × (perm join° check')) step ⊑ cons perm join° check' (6.6.3i)

if (x,b,y) is a checked triple for a permutation of l, then step(a,(x,b,y)) is a checked triple for a permutation of [a] ⧺ l

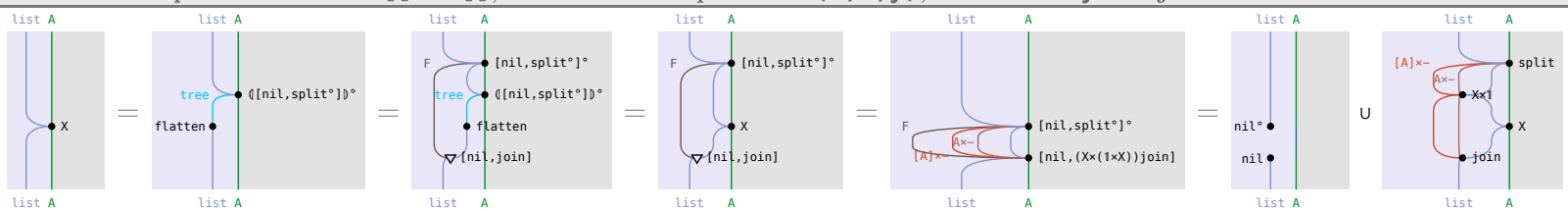


(1) (2)

1	(𝟙 × (perm join° check')) step	aRb puts a in front of x; otherwise bRa and a goes in front of y
2	⊑ cons perm join° check'	

X = ⦇[nil,split°]⦈° flatten ⟹ X = (nil° nil) ∪ (split (X × (𝟙 × X)) join) (6.6.3j)

quicksort returns [] on [], and otherwise splits into (x,a,y), sorts x and y and joins them around a

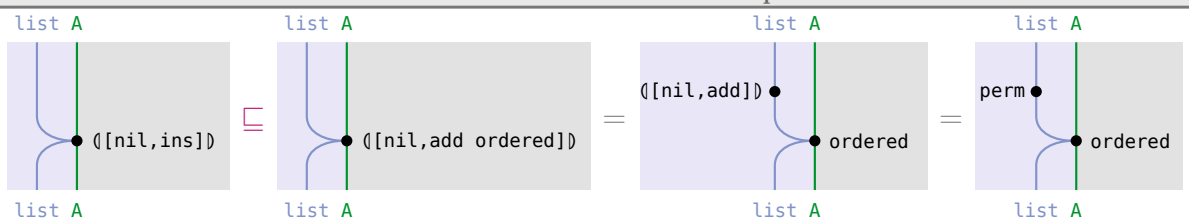


(1) (2) (3) (4) (5) (6)

1	X	X = ⦇[nil,split°]⦈° flatten
2	= ⦇[nil,split°]⦈° flatten	S° F(⦇S⦈° ⦇R⦈) R = ⦇S⦈° ⦇R⦈
3	= [nil,split°]° F(⦇[nil,split°]⦈° flatten) [nil,join]	X = ⦇[nil,split°]⦈° flatten
4	= [nil,split°]° F(X) [nil,join]	F(S) [T,U] = [T,(S × (𝟙 × S)) U]
5	= [nil,split°]° [nil,(X × (𝟙 × X)) join]	[U,V]° [R,S] = (U° R) ∪ (V° S)
6	= (nil° nil) ∪ (split (X × (𝟙 × X)) join)	

(𝟙 × ordered) ins ⊑ add ordered ⟹ ⦇[nil,ins]⦈ ⊑ perm ordered (6.6.3k)

for any ins that, on an ordered list, returns only ordered results of add, folding with [nil, ins] gives only sorted permutations

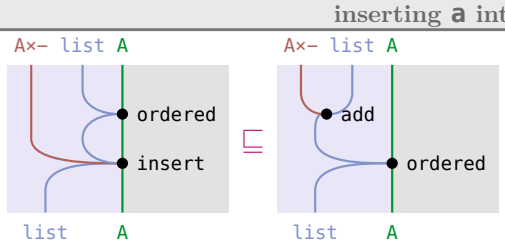


(1) (2) (3) (4)

1	⦇[nil,ins]⦈	(𝟙 × ordered) ins ⊑ add ordered ⟹ ⦇[nil,ins]⦈ ⊑ ⦇[nil,add ordered]⦈
2	⊑ ⦇[nil,add ordered]⦈	⦇[nil,add]⦈ ordered = ⦇[nil,add ordered]⦈
3	= ⦇[nil,add]⦈ ordered	perm = ⦇[nil,add]⦈
4	= perm ordered	

connected(R) ∧ (∀a. ∀b. ∀c. R(a,b) ⟹ R(b,c) ⟹ R(a,c)) ⟹ (𝟙 × ordered) insert ⊑ add ordered (6.6.3l)

inserting a into an ordered list puts a somewhere in it and keeps it ordered



(1) (2)

1	(𝟙 × ordered) insert	insert splices a in, and R transitive and connected
2	⊑ add ordered	

§6.7. Closure

$$\mathbf{R^* = (\mu X : \mathbb{1} \cup (X\ R))} \tag{6.7a}$$

closing **R** by composing it on the left and closing it by composing on the right give the same relation **R***

$$\mathbf{\mathbb{1} \sqsubseteq X \wedge X\ X \sqsubseteq X \Longrightarrow R \sqsubseteq X \Longleftrightarrow R^* \sqsubseteq X} \tag{6.7b}$$

X contains **R** exactly when it contains **R***

$$\mathbf{\mathbb{1} \sqsubseteq R^*} \tag{6.7c}$$

R* is reflexive

1	$\mathbb{1}$	$R \sqsubseteq R \cup S$
2	$\sqsubseteq \mathbb{1} \cup (R\ R^*)$	$\mathbb{1} \cup (R\ R^*) = R^*$
3	$= R^*$	

$$\mathbf{R \sqsubseteq R^*} \tag{6.7d}$$

R* contains **R**

1	R	
2	$= R$	(6.7c) $\mathbb{1} \sqsubseteq R^*$
3	$\sqsubseteq R\ R^*$	$R\ R^* \sqsubseteq R^*$
4	$\sqsubseteq R^*$	

$$\mathbf{(\mathbb{1} \cup (R\ (R^* / R^*)))\ R^* \sqsubseteq R^*} \tag{6.7e}$$

R*/R* is a prefixed point of **X ↦ 𝟙∪RX**, stated through division

$$\mathbf{R^*\ R^* \sqsubseteq R^*}$$

R* is transitive, because **R*/R*** is a prefixed point of **X ↦ 𝟙∪RX**

1	$(\mathbb{1} \cup (R\ (R^* / R^*)))\ R^*$	$(S \cup T)\ R = (S\ R) \cup (T\ R)$
2	$= (R^*) \cup (R\ (R^* / R^*)\ R^*)$	
3	$= R^* \cup (R\ (R^* / R^*)\ R^*)$	$(R / R)\ R \sqsubseteq R$
4	$\sqsubseteq R^* \cup (R\ R^*)$	$R\ R^* \sqsubseteq R^*$
5	$\sqsubseteq R^*$	

$$\mathbf{\mathbb{1} \sqsubseteq X \wedge X\ X \sqsubseteq X \wedge R \sqsubseteq X \Longrightarrow \mathbb{1} \cup (R\ X) \sqsubseteq X} \tag{6.7f}$$

a reflexive transitive **X** containing **R** is a prefixed point of **Y ↦ 𝟙∪RY**

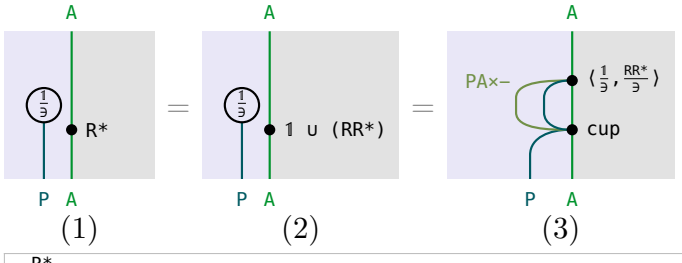
$$\mathbf{\mathbb{1} \sqsubseteq X \wedge X\ X \sqsubseteq X \wedge R \sqsubseteq X \Longrightarrow R^* \sqsubseteq X}$$

a reflexive transitive **X** containing **R** is a prefixed point of **X ↦ 𝟙∪RX**, so it contains **R***

1	$\mathbb{1} \cup (R\ X)$	$\mathbb{1} \sqsubseteq X$
2	$\sqsubseteq X \cup (R\ X)$	$R \sqsubseteq X$
3	$\sqsubseteq X \cup (X\ X)$	
4	$\sqsubseteq X$	

$$\mathbf{\frac{R^*}{\ni} = \langle \frac{\mathbb{1}}{\ni}, \frac{R\ R^*}{\ni} \rangle\ cup} \tag{6.7g}$$

the set of **R***-successors of **a** is **a** itself joined with the **R***-successors of its **R**-successors



(1) (2) (3)

1	$\frac{R^*}{\ni}$	$\mathbb{1} \cup (R\ R^*) = R^*$
2	$= \frac{\mathbb{1} \cup (R\ R^*)}{\ni}$	$\frac{R \cup S}{\ni} = \langle \frac{R}{\ni}, \frac{S}{\ni} \rangle\ cup$
3	$= \langle \frac{\mathbb{1}}{\ni}, \frac{R\ R^*}{\ni} \rangle\ cup$	

§6.7.1. Computing closure

$$R - \emptyset = R \tag{6.7.1a}$$

taking nothing away leaves R

$$R \cup (S - R) = R \cup S$$

beside R, only the part of S outside R adds anything

$$R - (S \cup T) = R - S - T$$

taking away S and then T takes away S∪T

$$(R \cup S) - T = (R - T) \cup (S - T)$$

taking T away from a union takes it away from each part

$$\text{monotonic}(\varphi) \wedge \text{monotonic}(\psi) \Longrightarrow (\mu X : \varphi(\psi(X))) = \varphi(\mu Y : \psi(\varphi(Y))) \tag{6.7.1b}$$

the least fixed point of φ after ψ is φ applied to the least fixed point of ψ after φ

$$S\ R^* = (\mu X : S \cup (X\ R)) \tag{6.7.1c}$$

S followed by any number of R steps is the least X containing S and closed under a further R step

$$R^*\ S = (\mu X : S \cup (R\ X))$$

any number of R steps followed by S is the least X containing S and closed under an R step in front

$$\theta(\emptyset,S) = S\ R^* \tag{6.7.1d}$$

started with nothing found and S to explore, θ computes SR*

1	$\theta(\emptyset,S)$	$\theta(P,Q) \triangleq P \cup (\mu X : Q \cup ((X\ R) - P))$
2	$\triangleq \emptyset \cup (\mu X : S \cup ((X\ R) - \emptyset))$	(6.7.1a) $R - \emptyset = R$
3	$= \emptyset \cup (\mu X : S \cup (X\ R))$	$\emptyset \cup R = R$
4	$= (\mu X : S \cup (X\ R))$	(6.7.1c) $S\ R^* = (\mu X : S \cup (X\ R))$
5	$= S\ R^*$	

$$\theta(P,\emptyset) = P \tag{6.7.1e}$$

with nothing left to explore, θ returns what it has found

1	$\theta(P,\emptyset)$	$\theta(P,Q) \triangleq P \cup (\mu X : Q \cup ((X\ R) - P))$
2	$\triangleq P \cup (\mu X : \emptyset \cup ((X\ R) - P))$	$(\mu X : \emptyset \cup ((X\ R) - P)) = \emptyset$
3	$= P \cup \emptyset$	$R \cup \emptyset = R$
4	$= P$	

§7. Optimisation Problems

§7.1. Minimum and maximum

(7.1a)

`est(R) ≜ ∋ ∩ (∈ \ R°)` — `x` is the `est(R)` of `xs` when `x∈xs` and `x R y` for every `y∈xs`; B&dM call it `min R`.

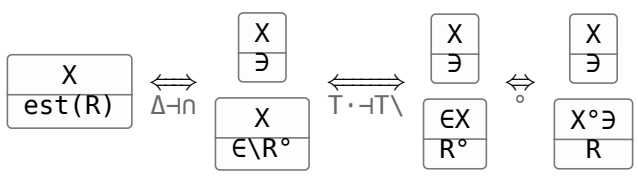
`est(≤) xs (est(≤)) x⟺x∈xs∧(∀y∈xs. x≤y).`

(7.1b)

	the law	what it says
1	`X⊑est(R)⟺X⊑∋` and `X°∋⊑R`	in the set, and below every element of it
2	`1/∋ (∈\R)=R`	bounding a singleton is bounding its element
3	`S/∋ (∈\R)=S°\R`	bound `S`'s image without building the set
4	`union ≜ ∋∋/∋`	flattens a set of sets
5	`union (∈\R)=∈\(∈\R)`	bound a union by bounding each member set
6	`1/∋ est(R) = 𝟙 ∩ R°` (7.4)	a singleton's minimum is its element, where `R` is reflexive `S/∋ est(R)` at `S:=𝟙`
7	`S/∋ est(R) = S ∩ (S° \ R°)`	an `S`-value that points to every `S`-value
8	`S/∋ est(R ∩ (S° S)) = S/∋ est(R)`	only `R` between values `S` gives one argument counts — context
9	`E(S) est(R)=(∋S)∩((∋S)°\R°)` (7.7)	the same for the image of a set `S/∋ est(R)` at `S:=∋S`
10	`Map(f) ⟹ P(f) est(R) = est(f R f°) f`	shunt a function through a minimum
11	`P(S) est(R)=(∋S)∩(∈\(SR°))` (7.9) `R` reflexive	fusion with the power relator `⊒` is the only proof here that tabulates
12	`P(S) est(R)⊑(∋S)∩(∈\(SR°))`	the half of the row above that costs nothing
13	`P(est(R)) est(R)⊑union est(R)` `R` transitive	a minimum in each set, then a minimum of those
14	`R R ⊑ R ⟹ P(est(R)) est(R) = P(dom(est(R))) union est(R)` (7.12)	the same as an equality, once empty sets are dropped

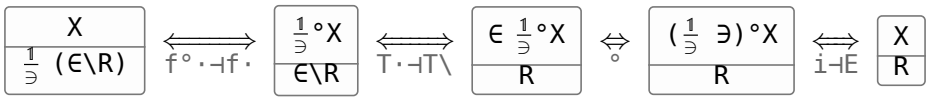
§7.1.1. `X ⊑ est(R) ⟺ X ⊑ ∋ ∧ ∈ X ⊑ R°` — X refines `est(R)` when it picks a member R-related to every member

(7.1.1a)



§7.1.2. `1/∋ (∈ \ R) = R` — all of {a} relates by R to c exactly when a does

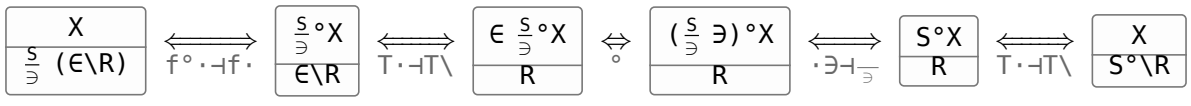
(7.1)



§7.1.3. `S/∋ (∈ \ R) = S° \ R` — gathering the S-values into one set first changes nothing

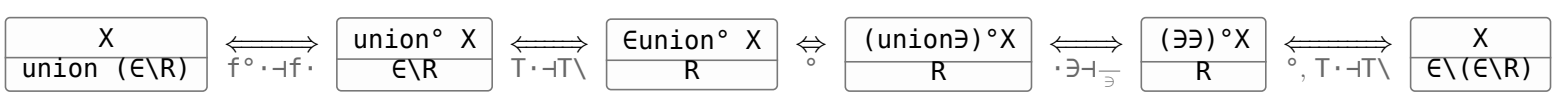
`S/∋` gathers the `S`-image of a point into one set and `∈\R` asks that every member of that set be `R`-related to the target, so the set cancels and `S°\R` asks it of the `S`-image directly.

(7.2)



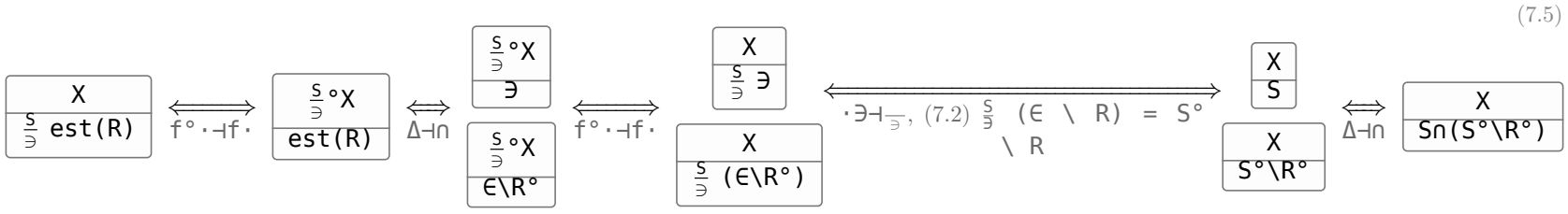
§7.1.4. `union (∈ \ R) = ∈ \ (∈ \ R)` — every member of the union relates by R iff every set's members do

(7.3)



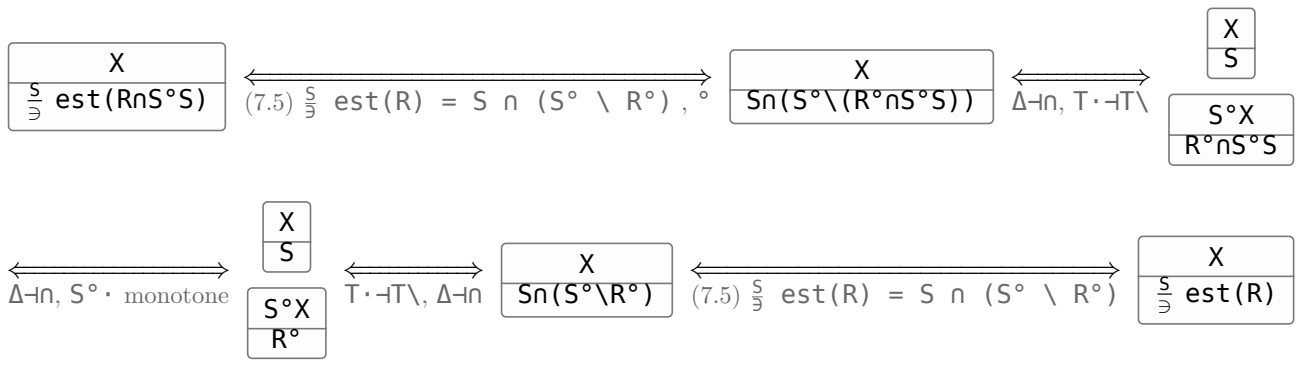
§7.1.5. `S/∋ est(R) = S ∩ (S° \ R°)` — the smallest S-value is an S-value R-related to every S-value

(7.5)



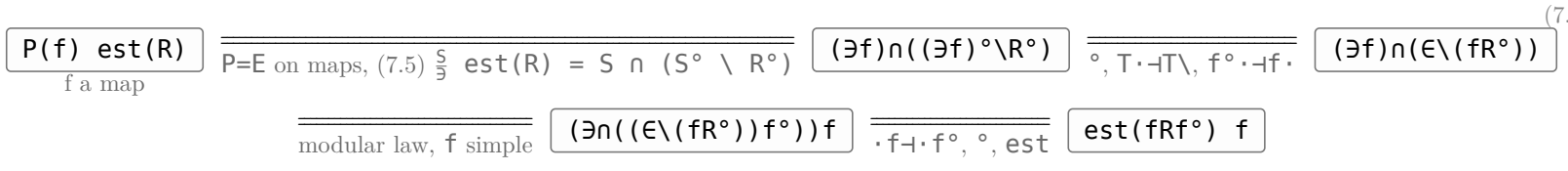
§7.1.6. $\frac{S}{\ni}$ est(R ∩ (S° S)) = $\frac{S}{\ni}$ est(R) — only R between S-values of one argument matters for the smallest

(7.6)



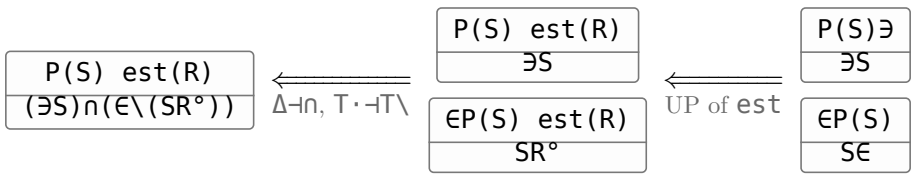
§7.1.7. Map(f) ⟹ P(f) est(R) = est(f R f°) f — the smallest f-image is f of the smallest under fRf°

(7.8)



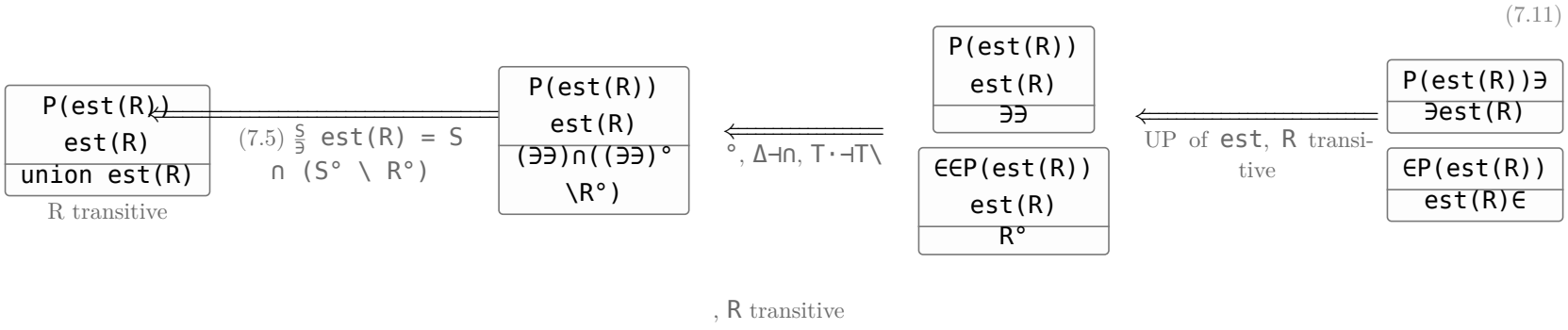
§7.1.8. P(S) est(R) ⊑ (∋ S) ∩ (∈ \ (S R°)) — the smallest of the S-image is R-below an S-value of every member

(7.10)



§7.1.9. R R ⊑ R ⟹ P(est(R)) est(R) ⊑ union est(R) — the smallest of the smallests is a smallest of the union

(7.11)



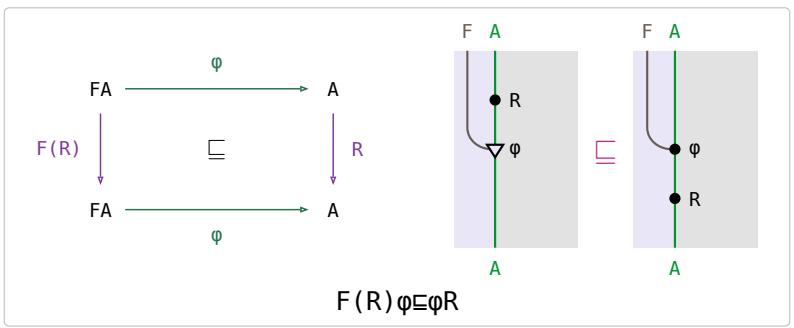
§7.2. Monotonic algebras

(7.2a)

φ Pres R ≜ F(R) φ ⊑ φ R — **φ preserves R**; B&dM call **φ monotonic on R**.

(≤×≤)+⊑+≤ — addition, ℕ×ℕ→ℕ , preserves ≤: c=a+b∧a≤a'∧b≤b'⟹c≤a'+b'.

(7.2b)

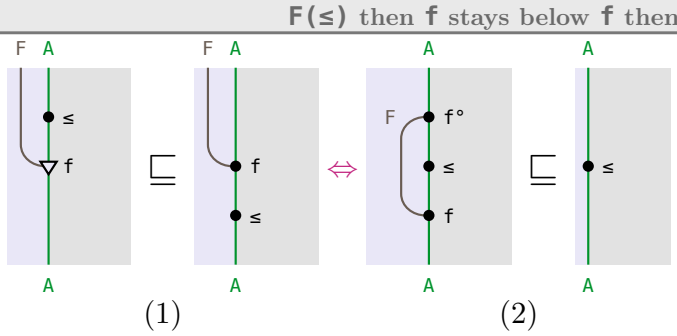


§7.2.1. f monotonic on ≤, read at points

(7.2.1a)

F(≤) f ⊑ f ≤ ⟺ f° F(≤) f ⊑ ≤

F(≤) then **f** stays below **f** then **≤** exactly when **f°**, **F(≤)**, **f** in a row stays below **≤**.



1	F(≤) f ⊑ f ≤	Map(f) ⟹ f° R ⊑ S ⟺ R ⊑ f S
2	⇔ f° F(≤) f ⊑ ≤	

$$\boxed{f(u)} \xrightarrow{f^\circ} \boxed{u} \xrightarrow{F(\leq)} \boxed{v} \xrightarrow{f} \boxed{f(v)}$$

- F(≤): from u to any v above it; u, v are in FA, compared component by component. This step is the premise.
- ⊑≤: the two ends satisfy f(u)≤f(v), the conclusion. So u F(≤) v ⟹ f(u)≤f(v): f preserves ≤.

§7.2.2. Function f is monotonic on R iff it distributes over R

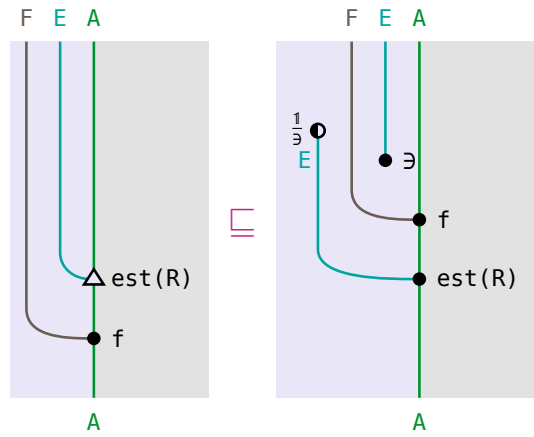
(7.2.2a)

f : FA→A **distributes over** R when F(est(R))f⊑ $\frac{F(\ni)f}{\ni}$ est(R).

(est(≤)×est(≤))+⊑ $\frac{(\ni\times\ni)+}{\ni}$ est(≤) — addition distributes over ≤: min(xs)+min(ys) is a least element of {x+y|x∈xs∧y∈ys}.

(7.2.2b)

$$\begin{tikzcd} F(PA) \arrow[r, "\frac{F(\ni)f}{\ni}"] \arrow[d, "F(est(R))"'] \arrow[dr, phantom, "\sqsubseteq"] & PA \arrow[d, "est(R)"] \\ FA \arrow[r, "f"'] & A \end{tikzcd}$$



$F(est(R))f \sqsubseteq \frac{F(\ni)f}{\ni}\ est(R)$

$$\begin{tikzcd} P\mathbb{N}\times P\mathbb{N} \arrow[r, "\frac{(\ni\times\ni)+}{\ni}"] \arrow[d, "est(\le)\times est(\le)"'] \arrow[dr, phantom, "\sqsubseteq"] & P\mathbb{N} \arrow[d, "est(\le)"] \\ \mathbb{N}\times\mathbb{N} \arrow[r, "+"'] & \mathbb{N} \end{tikzcd}$$

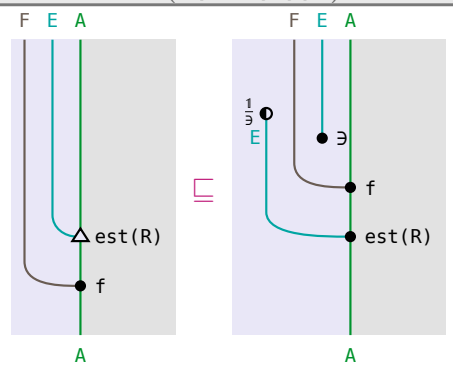
$(xs,ys) \mapsto \{x+y \mid x\in xs \wedge y\in ys\}$; $(min(xs),min(ys)) \mapsto min(xs)+min(ys)$

$(est(\le)\times est(\le))+ \sqsubseteq \frac{(\ni\times\ni)+}{\ni}\ est(\le)$

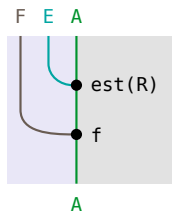
Theorem 7.1

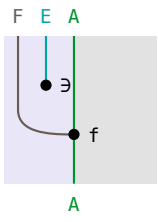
$$\mathbf{Map(f)} \wedge \mathbb{1} \sqsubseteq R \implies f^\circ\ F(R)\ f \sqsubseteq R \iff F(est(R))\ f \sqsubseteq \frac{F(\ni)\ f}{\ni}\ est(R)$$

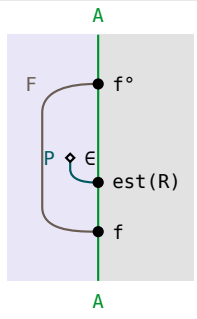
f carries **R**-related parts to **R**-related wholes (**f°F(R)f⊑R**) if and only if applying **f** to the **R**-best parts (**F(est(R))f**) is included in applying **f** to all parts, then taking an **R**-best (**F(∋)f** then **est(R)**)



f distributes over R — ≜ distributes — the fraction bent as $\frac{R}{\ni}\ \ni\ =\ R$

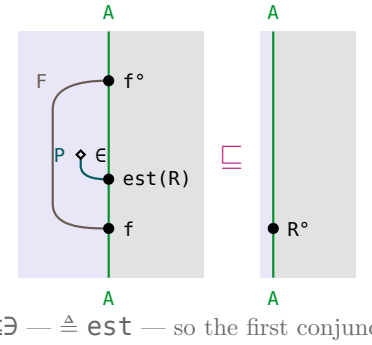
⇔ 

⊑ 

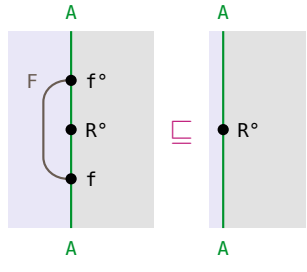
and 

⊑

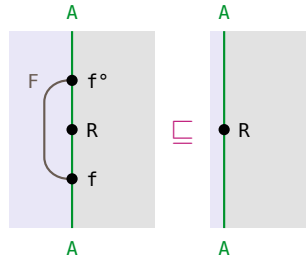
(7.5) $\frac{S}{\ni}\ est(R)\ =\ S \cap (S^\circ \setminus R^\circ)$ splits the bound in two, $X \sqsubseteq S \setminus R \iff S\ X \sqsubseteq R$ moving $(F(\ni)f)^\circ$ across

⇔ 

$est(R) \sqsubseteq \ni$ — ≜ est — so the first conjunct drops

⇔ 

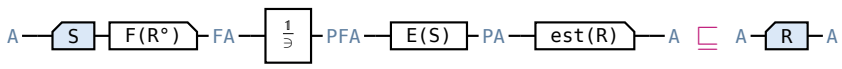
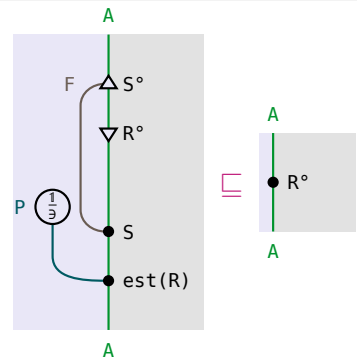
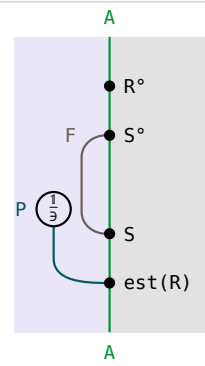
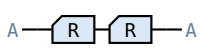
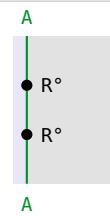
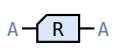
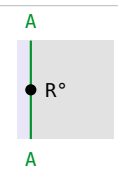
$(F(\ni)f)^\circ = f^\circ F(\in)$ — ≜ $^\circ$ — and $\in$ $est(R) = R^\circ$ — ≜ est, R reflexive

⇔ 

both sides conversed — $F(R^\circ)^\circ = F(R)$, $F(R^\circ)\ =\ F(R)^\circ$

§7.2.3. R R ⊑ R ∧ S Pres R ⟹ ⦇S/∋ est(R)⦈ ⊑ ⦇S⦈/∋ est(R) — taking the best at each step gives a best overall result

Theorem 7.2

circuit	Hinze–Marsden
⦇S/∋ est(R)⦈⊑⦇S⦈/∋ est(R) the conclusion: one minimum kept at each step is below every result collected and one minimum taken at the end	
⦇S/∋ est(R)⦈ ⊑ ⦇S⦈ and ⇔ ⦇S⦈°⦇S/∋ est(R)⦈ ⊑ R° (7.5) S/∋ est(R) = S ∩ (S° \ R°) at ⦇S⦈; ⦇S/∋ est(R)⦈⊑⦇S⦈°\R°⟺⦇S⦈°⦇S/∋ est(R)⦈⊑R° — X ⊑ S \ R ⟺ S X ⊑ R	
⟹ S/∋ est(R)⊑S the left conjunct; (7.5) S/∋ est(R) = S ∩ (S° \ R°) , and ⦇−⦈ is monotone	
 ⟹ the right conjunct; ⦇S⦈°⦇S/∋ est(R)⦈ is the least X with X=S°F(X)(S/∋ est(R)) hylomorphism theorem — so Knaster–Tarski leaves this one inequation S/∋ = 1/∋ E(S) — R/∋ ∋ = R	
⊑ S°F(R°)⊑R°S° — ≜ monotonic at S, conversed; F(R)°=F(R°) — F(R°) = F(R)°	
⊑  S°(S/∋ est(R))⊑S°(S°\R°)⊑R° — (7.5) S/∋ est(R) = S ∩ (S° \ R°) , (0.9a.4)	
⊑  R transitive	

§7.2.4. `Coreflexive(p) ⟹ takewhile(p) = ⦇[nil,(π₁p→cons,⊸ nil)]⦈` — `takewhile(p)` keeps items while `p` holds, as one fold

(7.2.4a)

	definition	type	example and in words
1	`FX = L + E × X`	$\mathcal{A}\to\mathcal{A}$	nothing, or a head and a tail
2	`nil ≜ []`	`1→[A]`	the empty list — the first constructor of `[A]`
3	`cons(a,[x₁,…,xₙ]) ≜ [a,x₁,…,xₙ]`	`A×[A]→[A]`	`cons(3,[1,2])=[3,1,2]` — a head onto a tail, the second constructor
4	`α ≜ [nil,cons]`	`F[A]→[A]`	both constructors as one map
5	`p ⊑ 𝟙`	`A→A`	`p≜even` — `2 p 2`, and `3∉dom(p)` — a coreflexive: `a` passes `p` when `a p a`
6	`R ≜ length ≤ length°`	`[A]→[A]`	`[1] R [1,2]` — `xs R ys⟺length(xs)≤length(ys)`
7	`(a,xs) ↦ []`	`A×[A]→[A]`	`(⊸ nil)(3,[1,2])=nil` — drop the pair, return `nil`
8	`prefix = ⦇[nil,cons ∪ ⊸ nil]⦈`	`[A]→[A]`	`[3,1,2] prefix [3,1]` — `xs prefix ys⟺∃zs. xs=ys⧺zs` — at each `cons`, keep the head or stop
9	`S ≜ [nil,((p × 𝟙) cons) ∪ ⊸ nil]`	`F[A]→[A]`	`(4,[2]) S [4,2]`, and `(3,[2]) S nil` only — `prefix`'s algebra with one extra `p` — keep a head that passes `p`, or stop
10	`(X→R,S) ≜ guard(C,X) [R,S]`	`A→B`	the test `X` picks the branch: `R` where `X` holds, `S` where it does not
11	`nil R=⊤, nil R°=nil`		`nil` is the shortest list — below every list, and above only itself, so it loses every `est(R°)`

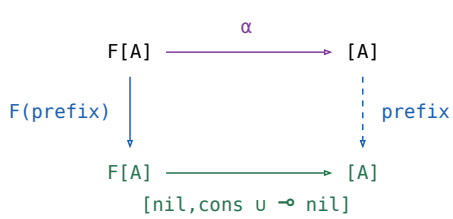
(7.2.4b)

`prefix = ⦇[nil,cons ∪ ⊸ nil]⦈`

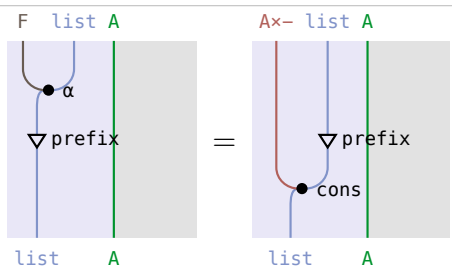
the fold whose algebra, at each `cons`, keeps the head or stops with `nil`: `xs prefix ys⟺∃zs. xs=ys⧺zs`

the defining square `α prefix=F(prefix)[nil,cons ∪ ⊸ nil]` — build the list and then take a prefix, or take a prefix of the tail and then rebuild with the algebra

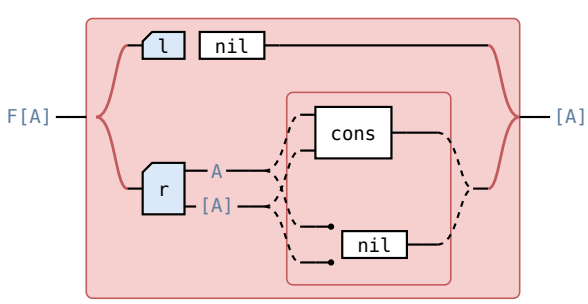
Hinze–Marsden



`F=1+A×−`, so `F(prefix)=𝟙+𝟙×prefix`: the head passes, the fold recurses on the tail alone



the `cons` operand of `cons ∪ ⊸ nil`; `⊸ nil` makes a constant and draws nothing



the algebra `[nil,cons ∪ ⊸ nil] : F([A])→[A]` as a circuit: `nil` on the `1` branch; on a pair `(a,ys´)` two outputs, `nil` and `cons(a,ys´)`

`nil prefix ys⟺ys=nil`
`cons(a,xs´) prefix ys⟺ys=nil ∨ ∃ys´. xs´ prefix ys´ ∧ ys=cons(a,ys´)`

a right fold: `α°` peels, `F(prefix)` recurses, the algebra runs on the way back; `init*`

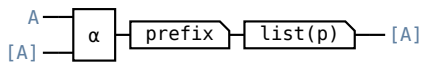
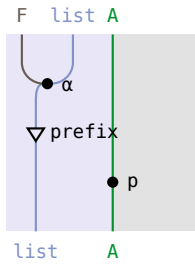
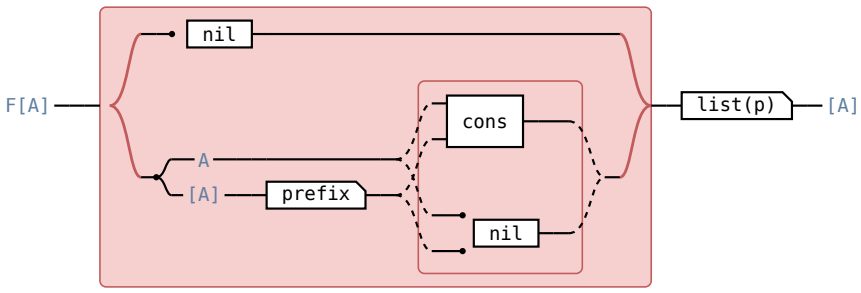
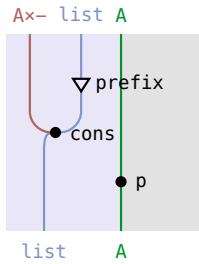
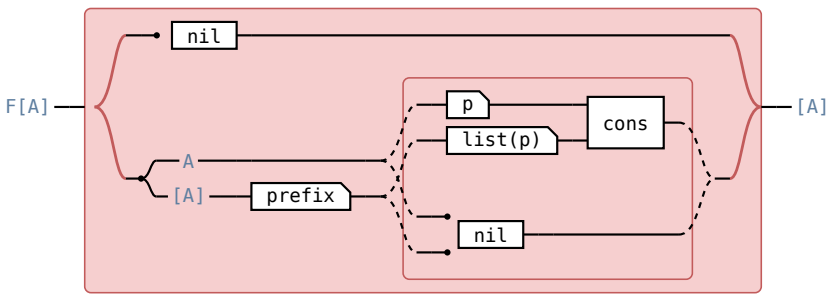
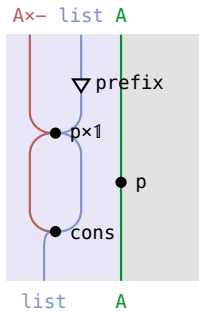
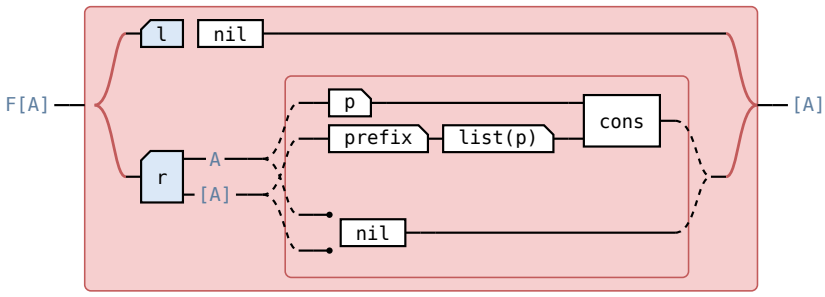
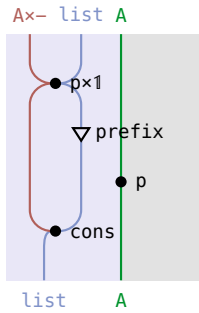
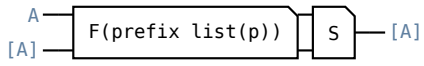
(`prefix = ⦇[nil,cons ∪ ⊸ nil]⦈` , `prefix = cat° π₁`) is the tail-recursive form

lax natural only, `list(R) prefix⊑prefix list(R)`: `list(R)` first needs an `R`-image of every element, `prefix` first may have dropped the ones without

Coreflexive(p) ⟹ α prefix list(p) = F(prefix list(p)) S (7.2.4c)

building the list and then keeping a **p**-passing prefix of it is keeping one of the tail first, and then building with **S**

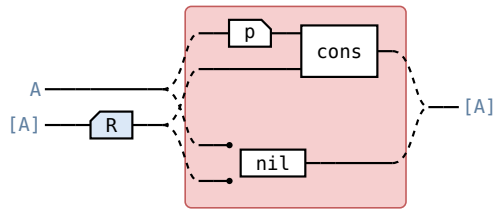
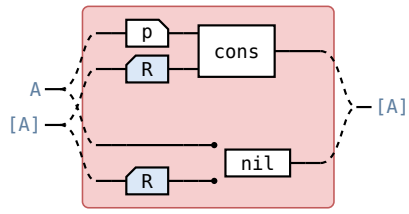
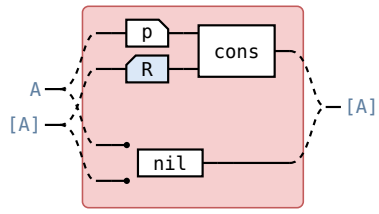
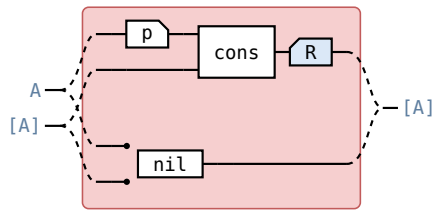
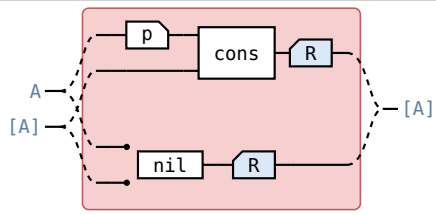
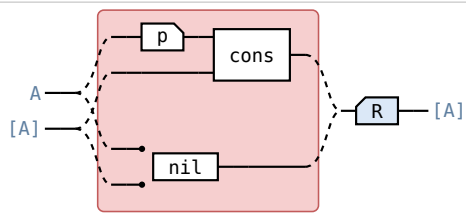
this same diagram is **subseq**'s: algebra **[nil,cons ∪ π₂]**, type **[A]→[A]**

circuit — the fork is F([A])=1+A×[A]: nil above, the pair below	**Hinze–Marsden**
 α prefix list(p)	 the cons branch alone, without 1+ or ⊸ nil
=  F(prefix) [nil,cons ∪ ⊸ nil] list(p) defining equation	 the cons operand of cons ∪ ⊸ nil
=  F(prefix) [nil,((p × list(p)) cons) ∪ ⊸ nil] list(p) through cons	 the (p×list(p)) cons operand of (p×list(p)) cons ∪ ⊸ nil
=  [nil,((p × (prefix list(p))) cons) ∪ ⊸ nil] relator, prefix entire	
=  F(prefix list(p)) S prefix list(p) entire	

≙ reduce reads that off as prefix list(p)=⦇S⦈. fusion cannot: list(p) is not entire, (𝟙×list(p))⊸ nil⊏⊸ nil, and no algebra meets the side condition. ,

Coreflexive(p) ⟹ (𝟙 × R°) (((p × 𝟙) cons) ∪ ⊸ nil) ⊑ (((p × 𝟙) cons) ∪ ⊸ nil) R° (7.2.4d)

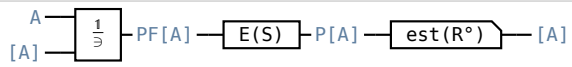
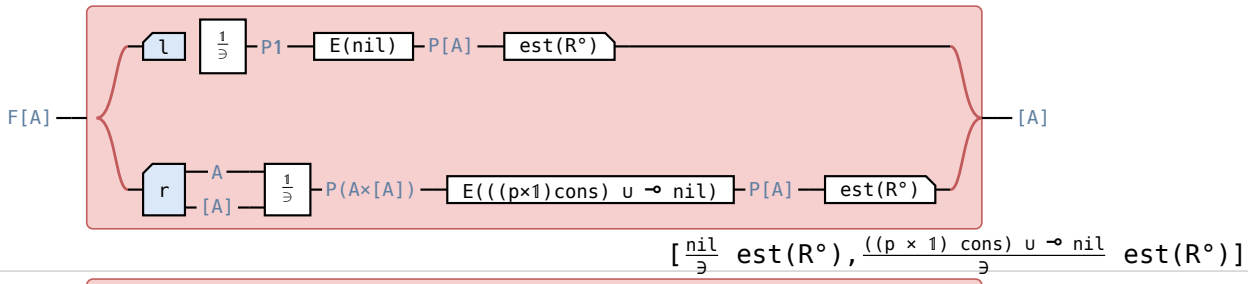
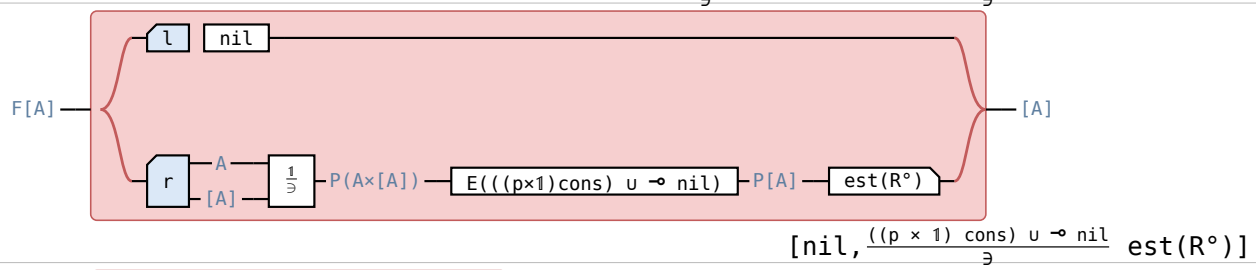
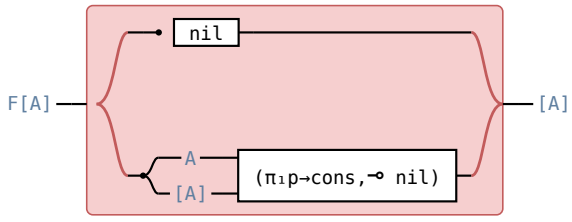
shortening the tail and then taking the step lands inside taking the step and then shortening the result

circuit — the cons branch of F(R°)S⊑SR°	**reason**
 (𝟙 × R°) (((p × 𝟙) cons) ∪ ⊸ nil)	
=  ((p × R°) cons) ∪ ((𝟙 × R°) ⊸ nil)	each operand is reached on its own (S ∪ T) R = (S R) ∪ (T R) , (R ∩ R') / S = (R / S) ∩ (R' / S) — and (𝟙×R°)(p×𝟙) is p and R° on the pair's two strands at once
⊑  ((p × R°) cons) ∪ ⊸ nil	⊸ is the greatest arrow into 𝟙, so (𝟙×R°)⊸⊑⊸
⊑  ((p × 𝟙) cons R°) ∪ ⊸ nil	cons length=(𝟙×length)π₂ succ with succ monotone — a shorter tail makes a shorter list
=  ((p × 𝟙) cons R°) ∪ (⊸ nil R°)	nil R°=nil (7.2.4a.11) — so the constant branch may carry the R° the other one already has
=  (((p × 𝟙) cons) ∪ ⊸ nil) R°	one R° past the join is the two inside it (S ∪ T) R = (S R) ∪ (T R) , (R ∩ R') / S = (R / S) ∩ (R' / S)

the nil branch, which no row above draws, is nil⊑nil R°.

Coreflexive(p) ⟹ $\frac{S}{\ni}$ est(R°) = [nil,(π₁p→cons,⊸ nil)] (7.2.4e)

the longest of the lists the algebra allows is the **cons** where the head passes **p**, and **nil** where it does not

formula	reason
	
= 	coproduct of maps
= 	singleton, R° reflexive
= 	nil R=⊤

the set is {nil} where p fails on the head and {nil,cons(a,xs)} where it holds, and nil loses the second — ≜ est at a two-element set.

(7.2.4f)

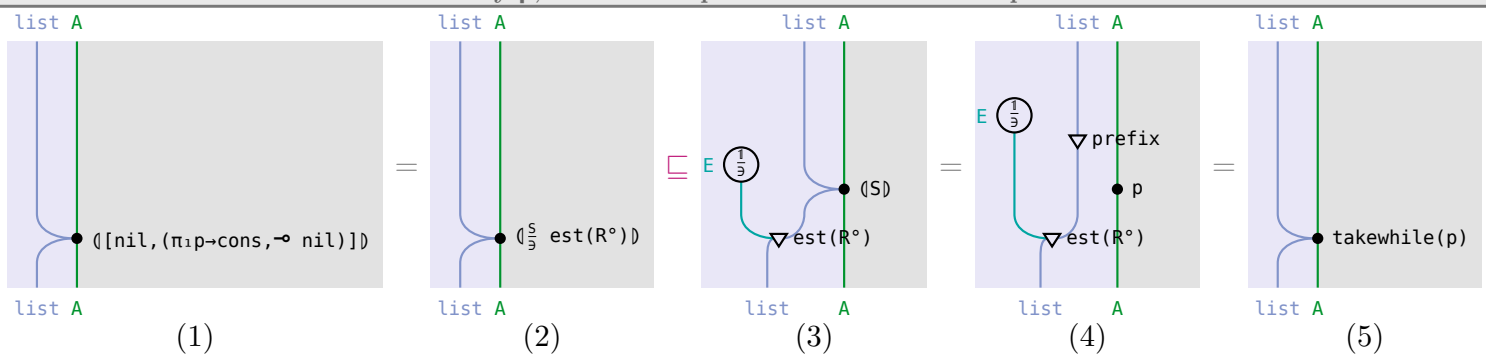
definition	type	example and in words
S ≜ [nil,((p × 𝟙) cons) ∪ ⊸ nil]	F[A]→[A]	(4,[2]) S [4,2], and (3,[2]) S nil only — prefix's algebra with one extra p — stop, or keep a head that passes p

Exercise 7.39

$$\texttt{takewhile(p)} \triangleq \frac{\texttt{prefix list(p)}}{\ni} \texttt{est(R°)}$$

$$\texttt{Coreflexive(p)} \implies \texttt{takewhile(p)} = \llparenthesis\texttt{[nil,(}\pi_1\texttt{p→cons,⊸ nil)]}\rrparenthesis$$

takewhile: `takewhile(p)(x)` returns the longest prefix of `x` with the property that all its elements satisfy `p`; the catamorphism is the standard implementation.



1	⦇[nil,(π₁p→cons,⊸ nil)]⦈	(7.2.4e) Coreflexive(p) $\implies \frac{S}{\ni}$ est(R°) = [nil,(π₁p→cons,⊸ nil)]
2	$= \llparenthesis \frac{S}{\ni}$ est(R°)⦈	Coreflexive(p) $\implies \llparenthesis \frac{S}{\ni}$ est(R°)⦈ $\sqsubseteq \frac{\llparenthesis S \rrparenthesis}{\ni}$ est(R°)
3	$\sqsubseteq \frac{\llparenthesis S \rrparenthesis}{\ni}$ est(R°)	Coreflexive(p) $\implies$ prefix list(p) = ⦇S⦈
4	$= \frac{\texttt{prefix list(p)}}{\ni}$ est(R°)	
5	= takewhile(p)	

takewhile(p)° takewhile(p)⊑prefix° prefix∩R∩R°⊑𝟙

takewhile(p)⊑prefix list(p) and (prefix list(p))° takewhile(p)⊑R — (7.5) $\frac{S}{\ni}$ est(R) = S ∩ (S° \ R°) at est(R°) — and two prefixes of one list of equal length are equal, so takewhile(p) is simple: **the** longest, not **a** longest.

$\frac{\texttt{prefix list(p)}}{\ni}$ est(R°) entire

nil is always a p-prefix and R is connected on the prefixes of one list, so the longest exists.

X⊑Y, X entire, Y simple ⟹X=Y

⦇[nil,(π₁p→cons,⊸ nil)]⦈ is a reduce of maps, hence entire — what turns the ⊒ above into the heading's =.

§7.2.5. mss = ⦇k⦈ π₂ est(≥) — mss is a fold followed by taking the largest

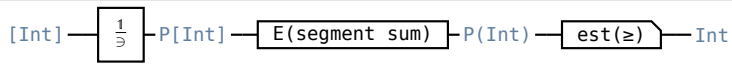
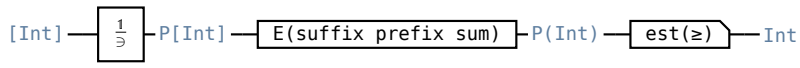
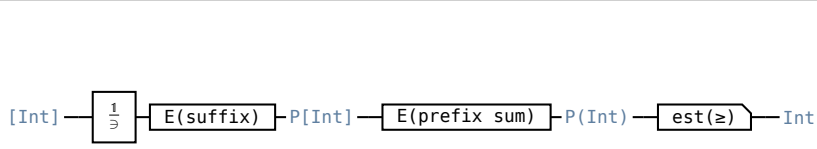
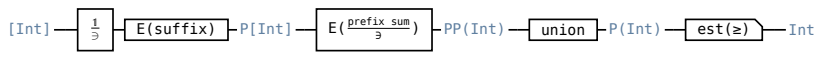
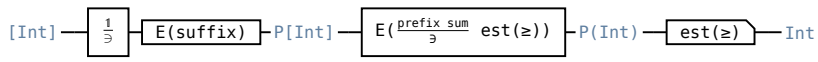
(7.2.5a)

$\oplus \triangleq \frac{\texttt{⊸ zero ∪ plus}}{\ni}$ est(≥) — ⊕ is the larger of 0 and a+b; B&dM write max(Λ(zero ∪ plus)).

$\oplus \triangleq \frac{\texttt{⊸ zero ∪ plus}}{\ni}$ est(≥) FX=1+Int×X, α≜[nil,cons], head≜cons° π₁, wrap≜⟨𝟙,⊸ nil⟩ cons.

mss = $\frac{\text{suffix}}{\ni}$ E($\frac{\text{prefix sum}}{\ni}$ est(≥)) est(≥) (7.2.5b)

maximum segment sum problem: using **segment=suffix prefix**, the specification is expressed in this form

formula — one wire from [Int] to Int, its type written along it	**reason**
	
= 	segment=suffix prefix segment = suffix prefix , ≜ mss
= 	absorption $\frac{S}{\ni}$ E(R) = $\frac{S\ R}{\ni}$ — frac(S,∋) E(R)=frac(SR,∋) at S:=suffix, R:=prefix sum
= 	$\frac{R}{\ni}$ ∋=R, union=E(∋) $\frac{R}{\ni}$ ∋ = R frac(R,∋)∋=R at R:=prefix sum and E(R)≜frac(∋R,∋), E(R) ≜ $\frac{\ni\ R}{\ni}$; union ≜ $\frac{\ni\ \ni}{\ni}$ union≜frac(∋∋,∋); the middle equality is ≜ relator's F(RS)=F(R)F(S) at F:=E
=	R R ⊑ R ⟹ P(est(R)) est(R) = P(dom(est(R))) union est(R) , the sets non-empty relator ≜ relator — F(RS)=F(R)F(S) at F:=E
= 	

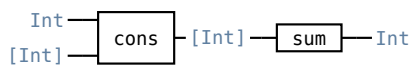
The union step is R R ⊑ R
⟹ P(est(R)) est(R) = P(dom(est(R))) union est(R) 's P(est(R)) est(R)=P(dom(est(R))) union est(R) — every suffix has the empty prefix, so Dom is 𝟙 here — and P(f)=E(f) at the map it is applied to (Map(f)
⟹ P(f) = E(f)).

[nil,cons ∪ ⊸ nil] sum = F(sum) S (7.2.5c)

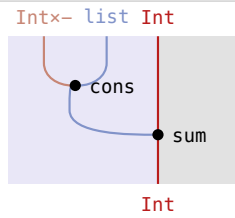
fusion: **prefix** expressed as a catamorphism on cons-lists, ⦇[nil,cons ∪ ⊸ nil]⦈, and this is the fusion condition that expresses **prefix sum** as a catamorphism

circuit — the fork is the bracket's case split
F([Int])=𝟙+Int×[Int]: nil above, the pair and its ∪ below

Hinze–Marsden

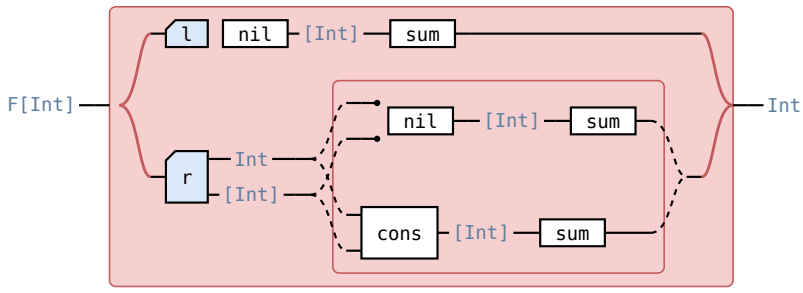


[nil,cons ∪ ⊸ nil] sum



the cons operand of cons ∪ ⊸ nil

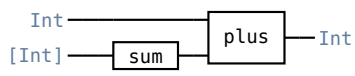
=



[nil sum,(⊸ nil sum) ∪ (cons sum)]

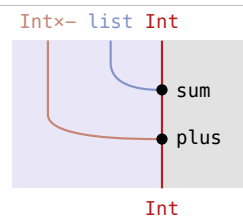
coproduct of maps, composition over ∪, ∪ commutative

=



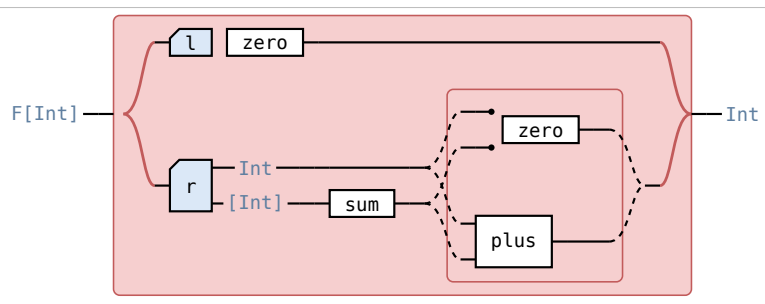
[zero,⊸ zero ∪ ((𝟙 × sum) plus)]

sum's defining equation



the (𝟙×sum) plus operand of ⊸ zero ∪ (𝟙×sum) plus

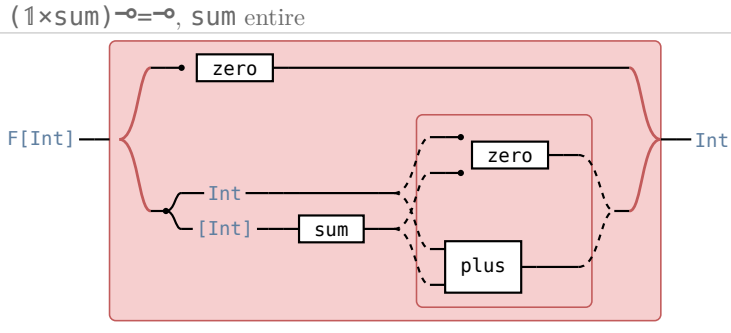
=



[zero,(𝟙 × sum) (⊸ zero ∪ plus)]

(𝟙×sum)⊸=⊸, sum entire

=



F(sum) [zero,⊸ zero ∪ plus]

relator

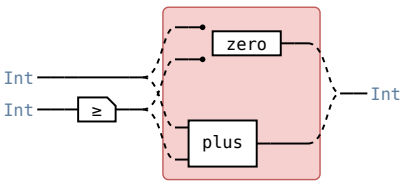
fusion at R:=[nil,cons ∪ ⊸ nil], S:=sum: the side condition, so prefix sum=⦇[zero,⊸ zero ∪ plus]⦈. prefix is the reduce, sum the map fused into it — the intermediate list is gone.

F(≥) S ⊑ S ≥ (7.2.5d)

monotonic algebra: an **F**-algebra **S** is monotonic on a relation **R** if **F(R)S⊑SR** — the **plus** branch of **F(≥)S⊑S≥**; the **zero** branch is **zero⊑zero≥**

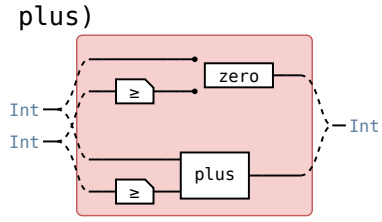
circuit — the head above, the running sum below; the tape is the ∪

(𝟙 × ≥) (⊸ zero ∪ plus)



=

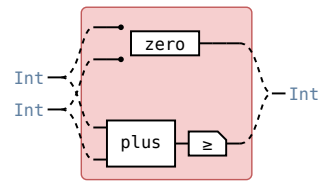
((𝟙 × ≥) ⊸ zero) ∪ ((𝟙 × ≥) plus)



relator, composition over ∪

⊑

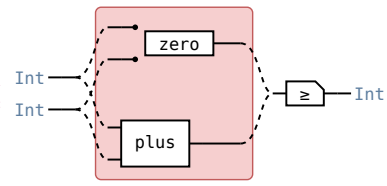
⊸ zero ∪ (plus ≥)



dom(R S) ⊑ dom(R) , (≥×≥) plus⊑plus≥; (≤×≤) plus⊑plus≤ is ≜ monotonic, written + there, and plus is a map, so it is monotonic on an order and on its opposite together, which carries it to ≥.

⊑

(⊸ zero ∪ plus) ≥

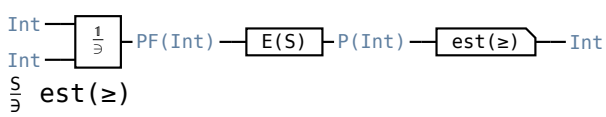


≥ reflexive

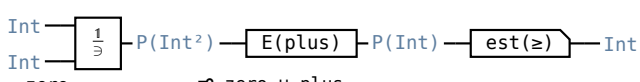
$\frac{S}{\ni}$ est(≥) = [zero,⊕] (7.2.5e)

the largest sum the algebra offers is zero from nothing and, from a head and a running sum, the larger of zero and the head added to it

circuit — the tape is the coproduct: zero's branch above, plus's below



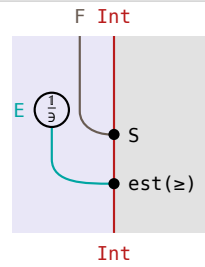
=



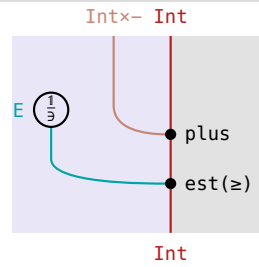
[zero/∋ est(≥), (⊸ zero ∪ plus)/∋ est(≥)]

coproduct of maps — [R,S] = [R/∋,S/∋] ∋ at T:=[zero,⊸ zero ∪ plus], then [U,V]Z=[UZ,VZ] — [R,S] Z = [R Z,S Z] , composition over ∪

Hinze–Marsden

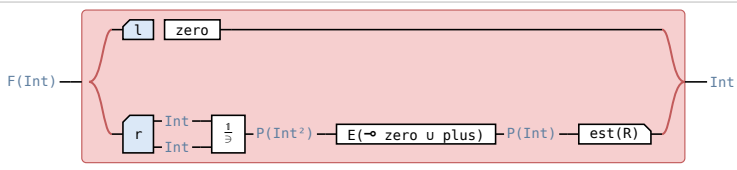


the zero arm of the bracket

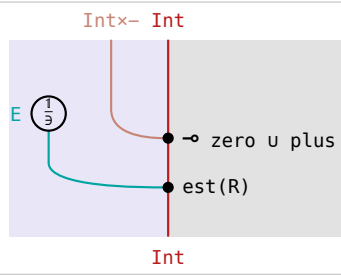


the plus operand of the lower arm's ⊸ zero ∪ plus, under its $\frac{1}{\ni}$ E(…) and est(≥)

=



singleton, ≥ reflexive — $\frac{1}{\ni}$ est(R) = 𝟙 ∩ R° 's $\frac{1}{\ni}$ est(R)=𝟙∩R at R:=≥, zero a map; the lower branch is ⊕'s definition, ≜ mss, and no law



with F(≥) S ⊑ S ≥ the greedy theorem gives ⦇[zero,⊕]⦈⊑ $\frac{\text{prefix sum}}{\ni}$ est(≥) — and Exercise 7.40 mss = ⦇k⦈ π₂ est(≥) makes it an equality.

$$[\mathsf{nil},\mathsf{cons}]\ (\!|[\mathsf{zero},\oplus]|\!),\tfrac{\mathsf{suffix}}{\ni}\ \mathsf{E}(\!|[\mathsf{zero},\oplus]|\!)) = \mathsf{F}((\!|[\mathsf{zero},\oplus]|\!),\tfrac{\mathsf{suffix}}{\ni}\ \mathsf{E}(\!|[\mathsf{zero},\oplus]|\!)))\ \mathsf{k} \qquad (7.2.5f)$$

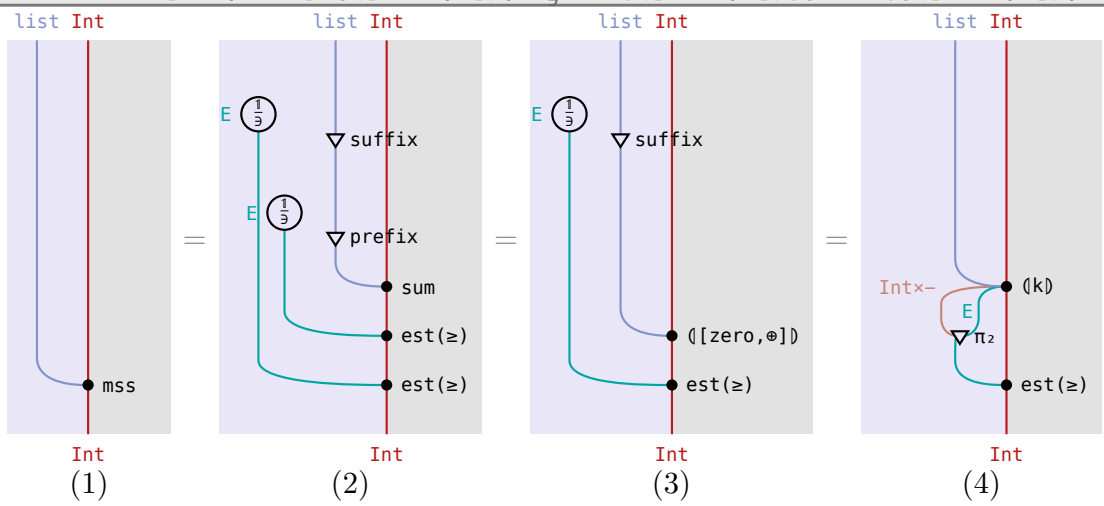
$\mathsf{k} \triangleq [\mathsf{zero}\langle \mathbb{1},\tfrac{\mathbb{1}}{\ni}\rangle,\langle \mathsf{w},\langle \mathsf{w}\ \tfrac{\mathbb{1}}{\ni},\pi_2\pi_2\rangle\ \mathsf{cup}\rangle]$, $\mathsf{w} \triangleq (\mathbb{1}\times\pi_1)\oplus$: the value at the whole list, paired with the set of the values at its suffixes, runs k's recursion.

the equation at that branch	why
$\mathsf{nil}\langle \mathsf{g},\tfrac{\mathsf{suffix}}{\ni}\ \mathsf{E}(\mathsf{g})\rangle = \mathsf{zero}\langle \mathbb{1},\tfrac{\mathbb{1}}{\ni}\rangle$	`nil` has one suffix, itself, and `g(nil)=zero`, so the set is the singleton `{zero}`
$\mathsf{cons}\langle \mathsf{g},\tfrac{\mathsf{suffix}}{\ni}\ \mathsf{E}(\mathsf{g})\rangle = (\mathbb{1}\times\langle \mathsf{g},\tfrac{\mathsf{suffix}}{\ni}\ \mathsf{E}(\mathsf{g})\rangle)\langle \mathsf{w},\langle \mathsf{w}\ \tfrac{\mathbb{1}}{\ni},\pi_2\pi_2\rangle\ \mathsf{cup}\rangle$	`g(cons(a,x))=a⊕(g(x))`, which is `w` reading `g(x)` off π_1; and the suffixes of `cons(a,x)` are `cons(a,x)` itself, whose value is that same `w`, together with those of `x`, which $\pi_2\pi_2$ carries — so the two sets meet at `cup`

$$\mathsf{mss} = (\!|\mathsf{k}|\!)\ \pi_2\ \mathsf{est}(\geq)$$

Exercise 7.40

maximum segment sum problem: $\tfrac{\mathsf{suffix}}{\ni}\ \mathsf{E}(\!|[\mathsf{zero},\oplus]|\!))$ expressed as the catamorphism $(\!|\mathsf{k}|\!)$, hence mss implemented by a linear-time algorithm, $\oplus \triangleq \tfrac{\multimap\ \mathsf{zero}\ \cup\ \mathsf{plus}}{\ni}\ \mathsf{est}(\geq)$ — $\triangleq \mathsf{mss}$; k and w — $[\mathsf{nil},\mathsf{cons}]\ (\!|[\mathsf{zero},\oplus]|\!),\tfrac{\mathsf{suffix}}{\ni}\ \mathsf{E}(\!|[\mathsf{zero},\oplus]|\!)) = \mathsf{F}((\!|[\mathsf{zero},\oplus]|\!),\tfrac{\mathsf{suffix}}{\ni}\ \mathsf{E}(\!|[\mathsf{zero},\oplus]|\!)))\ \mathsf{k}$.



1	mss	(7.2.5b) $\mathsf{mss} = \tfrac{\mathsf{suffix}}{\ni}\ \mathsf{E}(\tfrac{\mathsf{prefix\ sum}}{\ni}\ \mathsf{est}(\geq))\ \mathsf{est}(\geq)$
2	$= \tfrac{\mathsf{suffix}}{\ni}\ \mathsf{E}(\tfrac{\mathsf{prefix\ sum}}{\ni}\ \mathsf{est}(\geq))\ \mathsf{est}(\geq)$	$\tfrac{\mathsf{prefix\ sum}}{\ni}\ \mathsf{est}(\geq) = (\!|[\mathsf{zero},\oplus]|\!)$
3	$= \tfrac{\mathsf{suffix}}{\ni}\ \mathsf{E}(\!|[\mathsf{zero},\oplus]|\!))\ \mathsf{est}(\geq)$	$(\!|\mathsf{k}|\!)\ \pi_2 = \tfrac{\mathsf{suffix}}{\ni}\ \mathsf{E}(\!|[\mathsf{zero},\oplus]|\!))$
4	$= (\!|\mathsf{k}|\!)\ \pi_2\ \mathsf{est}(\geq)$	

one fold builds the `n+1` running maxima and the final `est(≥)` reads them in one more pass, so `mss` is linear.

§7.2.6. Coreflexive(p) ⟹ filter(p) = ⦇[nil,(π₁p→cons,π₂)]⦈ — filter(p) keeps the items satisfying p, as one fold

(7.2.6a)

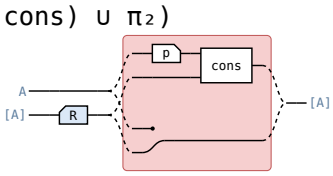
	definition	type	example and in words
1	FX = L + E × X	$\mathcal{A}\to\mathcal{A}$	nothing, or a head and a tail
2	α ≜ [nil,cons]	F[A]→[A]	both constructors as one map
3	p ⊑ 𝟙	A→A	p≜even — 2 p 2, and 3∉dom(p) — a coreflexive: a passes p when a p a
4	R ≜ length ≤ length°	[A]→[A]	[1] R [1,2] — xs R ys⟺length(xs)≤length(ys)
5	π₂(a,x) ≜ x	A×[A]→[A]	π₂(3,[1,2])=[1,2] — drop the head, keep the tail
6	subseq = ⦇[nil,cons ∪ π₂]⦈	[A]→[A]	[3,1,2] subseq [3,2] — xs subseq ys⟺ys is xs with elements dropped — at each cons, keep the head or drop it
7	S ≜ [nil,((p × 𝟙) cons) ∪ π₂]	F[A]→[A]	(4,[2]) S [4,2] and (4,[2]) S [2], but (3,[2]) S [2] only — subseq's algebra with one extra p — keep a head that passes p, or drop the head
8	𝟙⊑π₂R cons°		the tail is one shorter than the cons, so π₂ loses the est(R°) at every step — where (a,xs) ↦ [] 's loser is nil

Coreflexive(p) ⟹ F(R°) S ⊑ S R° (7.2.6b)

shortening the tail and then taking the step lands inside taking the step and then shortening the result

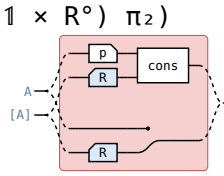
formula — the cons branch of F(R°)S⊑SR°; **reason** under each circuit

(𝟙 × R°) (((p × 𝟙) cons) ∪ π₂)



=

((p × R°) cons) ∪ ((𝟙 × R°) π₂)

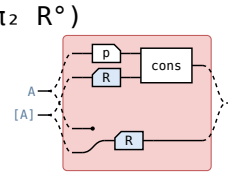


(𝟙×R°)(p×𝟙)=p×R° ≜ (𝟙×R°)π₂=π₂R° (𝟙 × ∋) π₂ = π₂ ∋

relator

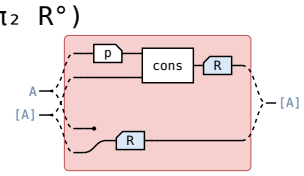
=

((p × R°) cons) ∪ (π₂ R°)



⊑

((p × 𝟙) cons R°) ∪ (π₂ R°)

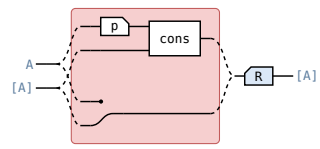


(p×R°) cons⊑(p×𝟙) cons R°

Coreflexive(p) ⟹ (𝟙 × R°) (((p × 𝟙) cons) ∪ ⊸ nil) ⊑ (((p × 𝟙) cons) ∪ ⊸ nil) R°

⊑

(((p × 𝟙) cons) ∪ π₂) R°

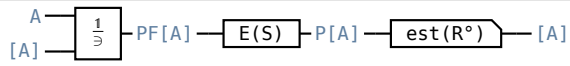
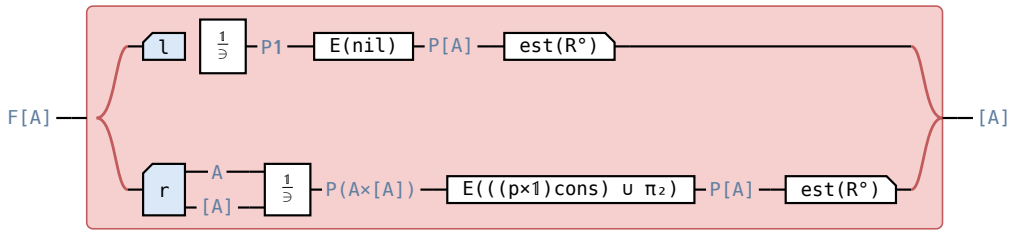
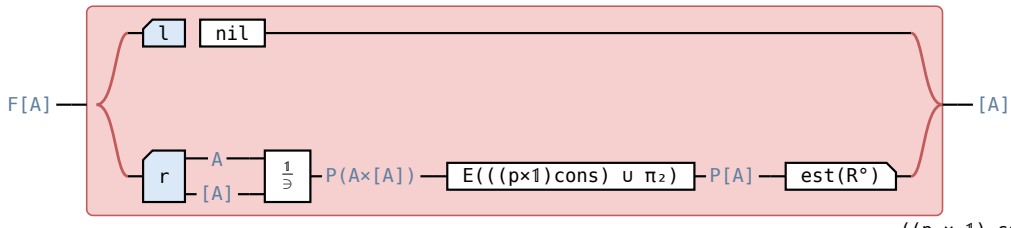
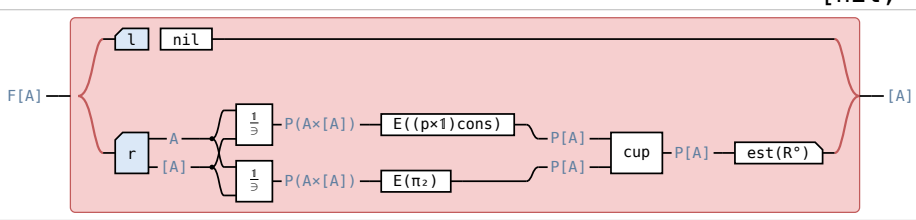
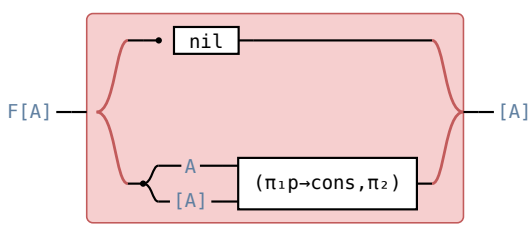


(S ∪ T) R = (S R) ∪ (T R) , (R ∩ R') / S = (R / S) ∩ (R' / S)

the nil branch: nil⊑nil R°.

Coreflexive(p) ⟹ $\frac{S}{\ni}$ est(R°) = [nil,(π₁p→cons,π₂)] (7.2.6c)

the longest of the lists the algebra allows is the **cons** where the head passes **p**, and the tail where it does not

formula	reason
	
=  $[\frac{nil}{\ni}\ est(R°), \frac{((p \times 1)\ cons) \cup \pi_2}{\ni}\ est(R°)]$	S ≜ [nil,((p × 𝟙) cons) ∪ π₂] — and the fraction of a coproduct of maps is the coproduct of their fractions [R,S] = $[\frac{R}{\ni},\frac{S}{\ni}]$ ∋
=  $[nil, \frac{((p \times 1)\ cons) \cup \pi_2}{\ni}\ est(R°)]$	$\frac{nil}{\ni}$ is the singleton {nil}, and est(R°) of a singleton is its element because R° is reflexive ≜ est
= 	$\frac{(p\times 1)\ cons \cup \pi_2}{\ni}$ $=\langle \frac{(p\times 1)\ cons}{\ni}, \frac{\pi_2}{\ni} \rangle$ cup ≜ cup
= 	𝟙⊑π₂R cons° (7.2.6a.8) — xs R cons(a,xs), so est(R°) returns the cons where p a puts it in the set and xs where the set is {xs} ≜ est

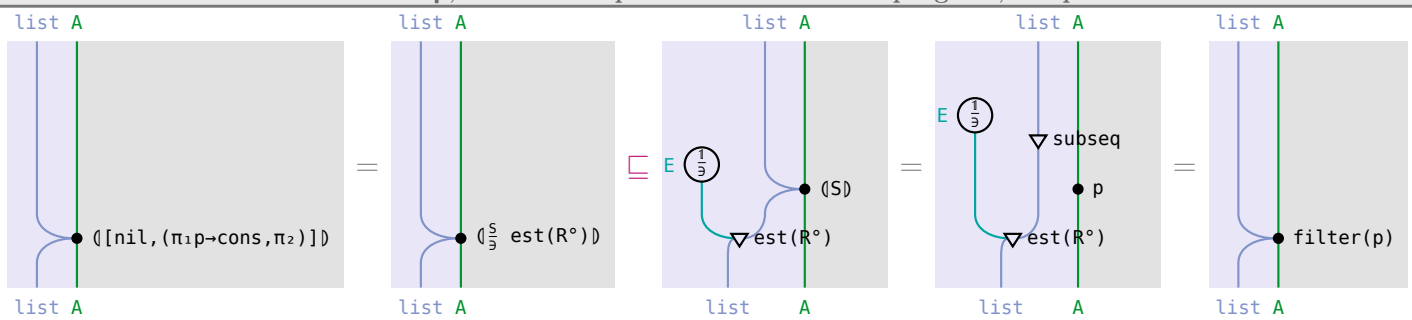
the head is dropped, not the whole tail: that is the one place π₂ shows against Coreflexive(p)
⟹ $\frac{S}{\ni}$ est(R°) = [nil,(π₁p→cons,⊸ nil)] 's ⊸ nil.

(7.2.6d)

definition	type	example and in words
S ≜ [nil,((p × 𝟙) cons) ∪ π₂]	F[A]→[A]	(4,[2]) S [4,2] and (4,[2]) S [2], but (3,[2]) S [2] only — subseq's algebra with one extra p — keep a head that passes p, or drop the head

Coreflexive(p) ⟹ filter(p) = ⦇[nil,(π₁p→cons,π₂)]⦈ Exercise 7.41

filter: **filter(p) x** returns the longest subsequence of **x** with the property that all its elements satisfy **p**; the catamorphism is the standard program; **R** a preorder.



(1) (2) (3) (4) (5)

1	⦇[nil,(π₁p→cons,π₂)]⦈	(7.2.6c) Coreflexive(p) ⟹ S/∋ est(R°) = [nil,(π₁p→cons,π₂)]
2	= ⦇S/∋ est(R°)⦈	Coreflexive(p) ⟹ ⦇S/∋ est(R°)⦈ ⊑ ⦇S⦈/∋ est(R°)
3	⊑ ⦇S⦈/∋ est(R°)	Coreflexive(p) ⟹ subseq list(p) = ⦇S⦈
4	= subseq list(p)/∋ est(R°)	
5	= filter(p)	

the catamorphism is entire and filter(p) simple, so ⊑ is =.

(subseq list(p))°(subseq list(p))∩R∩R°⊑𝟙 fails — two p-subsequences of one list can be of equal length and different — so §7.2.4's uniqueness argument does not transfer. What survives est(R°) is the one keeping **exactly** the passing elements: a subsequence that drops a passing element is beaten by the one that keeps it.

(7.2.6e)

	definition	type	note
1	segment = suffix prefix	[A]→[A]	A contiguous stretch of xs: a suffix, then a prefix of that.
2	filter(p)≜ subseq list(p)/∋ est(R°)	[A]→[A]	The longest subsequence of xs whose every element passes p. est(R°) is ≜ est
3	R≜length≤length°	[A]→[A]	The preorder filter and takewhile maximise over: the longer list wins. ≥≜≤°
4	takewhile(p)≜ prefix list(p)/∋ est(R°)	[A]→[A]	The same with prefix for subseq: the longest prefix whose every element passes p.
5	mss≜ segment sum/∋ est(≥)	[Int]→Int	Maximum segment sum. segment=suffix prefix splits it into prefix sum/∋ est(≥) on each suffix.

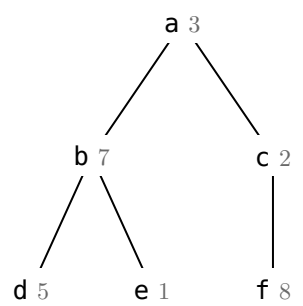
§7.3. Planning a company party

(7.3a)

	definition	type	note
1	`tree A::=node (A,[tree A])`	$\mathcal{A}\to\mathcal{A}$	The company hierarchy: an employee, and the list of subtrees under them.
2	`F(A,B)=A×[B]`	$\mathcal{A}\to\mathcal{A}$	The base functor `tree` folds: an employee beside the recursive position, one layer deep.
3	`rating`	`A→Int`	What one employee is worth as a guest.
4	`cost = list(rating) sum`	`[A]→Int`	What a guest list is worth.
5	`R = cost ≤ cost°`	`[A]→[A]`	The preorder the guest list is maximised over.
6	`choose ≜ π₁ ∪ π₂`	`[A]×[A]→[A]`	Takes one of the two parties a subtree returns.
7	`include = (𝟙 × (list(π₂) concat)) cons`	`F(A,[A]×[A])→[A]`	The party that invites the root, which puts every immediate subtree's root out. A map.
8	`exclude = (𝟙 × (list(choose) concat)) π₂`	`F(A,[A]×[A])→[A]`	The party that leaves the root out, so each subtree is free to choose. Not a map.
9	`S ≜ ⟨include,exclude⟩`	`F(A,[A]×[A])→[A]×[A]`	The algebra: one step returns both parties of a subtree at once.
10	`party = ⦇S⦈ choose`	`tree(A)→[A]`	Every guest list the president's ruling allows.
11	the specification $\frac{\texttt{party}}{\ni}$ `est(R°)`	`TA→[A]`	A guest list of greatest total conviviality.

§7.3.1. `include` and `exclude`

(7.3.1a)



the small grey number is the employee's rating

(7.3.1b)

	node	the algebra's input `A×[[A]×[A]]`	`include`	`exclude`
1	`d, e, f`	`(d,[])`	`[d]`	`[]`
2	`b`	`(b,[([d],[]),([e],[])])`	`[b]`	`[d,e], [d], [e], []`
3	`c`	`(c,[([f],[])])`	`[c]`	`[f], []`
4	`a`	`(a,[([b],…),([c],…)])`	`[a]⧺exclude(b)⧺exclude(c)`	`choose(b)⧺choose(c)`

with those ratings, `est(R°)` keeps `([b]=7,[d,e]=6)` at `b`, where `include` wins, and `([c]=2,[f]=8)` at `c`, where `exclude` wins, so at the root `include=[a,d,e,f]=17` beats `exclude=[b,f]=15`, and `choose` takes 17.

(7.3.1c)

(7.3.1d)

(7.3.1e)

```
list(choose) [([d],[]),([e],[])]
  1st item              ([d],[])  choose↦[d]  or  []
  2nd item              ([e],[])  choose↦[e]  or  []
  2×2=4 combinations:   [[d],[e]]   [[d],[]]   [[],[e]]   [[],[]]
  concat flattens:        [d,e]       [d]        [e]        []
```

§7.3.2. list((R×R)°)

(7.3.2a)

```
[  ([b],[d,e])  ,  ([c],[f])  ]
        |              |
        ▼              ▼        list((R×R)°), elementwise
[   ([b],[d])   ,  ([c],[])   ]

([b],[d,e])
  |   └── exclude: the best party in b's subtree when b does not come
  └────── include: the best party in b's subtree when b comes
```

(7.3.2b)

	element	1st component include	2nd component exclude
1	1st, ([b],[d,e])	[b] R° [b] 7≥7, reflexivity	[d,e] R° [d] 6≥5, and ([b],[d,e]) is the pair est(R°) keeps at b
2	2nd, ([c],[f])	[c] R° [c] 2≥2, reflexivity	[f] R° [] 8≥0, and ([c],[f]) is the pair est(R°) keeps at c

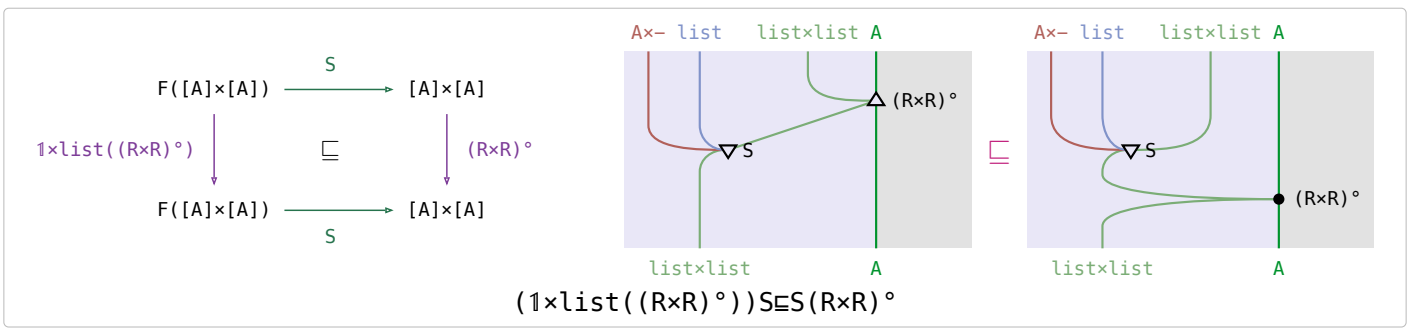
the first component of 𝟙×list((R×R)°) is the root employee.

R≜cost≤cost° is a preorder, so the components that do not move are related by reflexivity — (R×R)° demands a relation in **both** components, it cannot skip one.

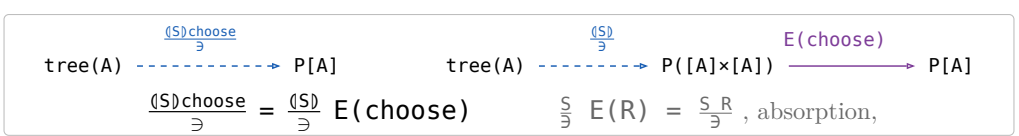
R compares cost only, not contents: [f] R° [] is legal though [f] has an element [] does not. That is why the three leaves of §7.3.3's proof all reduce to "cost is a sum".

§7.3.3. (𝟙 × list((R × R)°)) S ⊑ S (R × R)° — S is monotonic on (R×R)°

(7.3.3a)



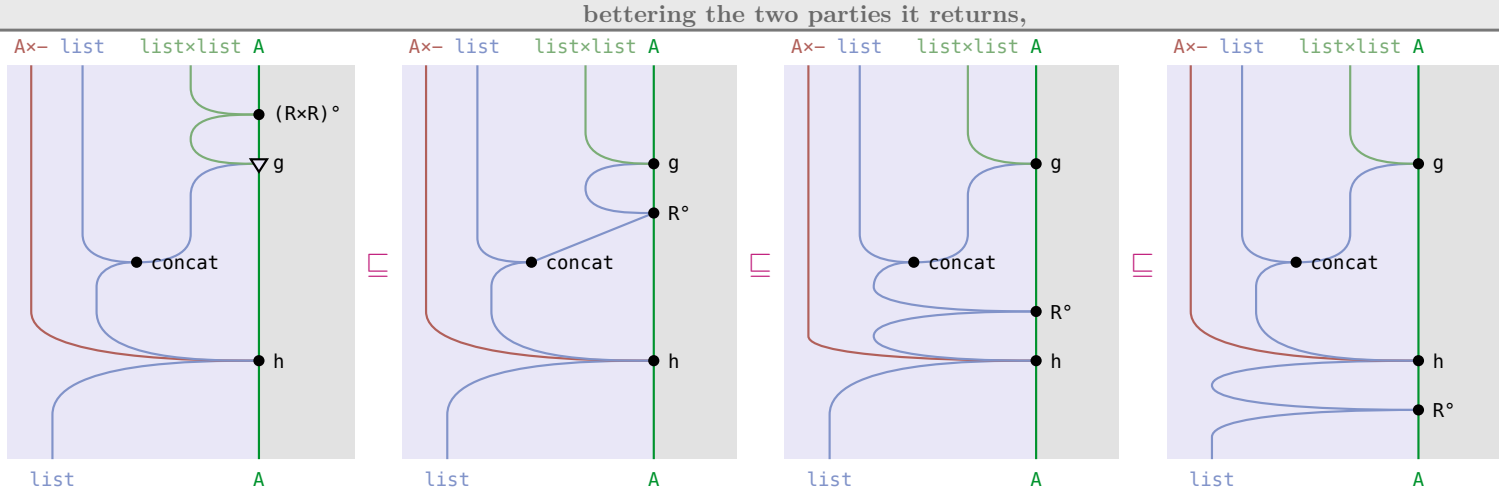
(7.3.3b)



(7.3.3c)

$$S \triangleq \langle \text{include}, \text{exclude} \rangle$$
$$(\mathbb{1} \times \text{list}((R \times R)^\circ))\ \text{include} \sqsubseteq \text{include}\ R^\circ$$
$$(\mathbb{1} \times \text{list}((R \times R)^\circ))\ \text{exclude} \sqsubseteq \text{exclude}\ R^\circ$$
$$(\mathbb{1} \times \text{list}((R \times R)^\circ))\ S \sqsubseteq S\ (R \times R)^\circ$$

bettering both parties of every subtree before the node's algebra runs gets no further than running it first and bettering the two parties it returns,



(1) (2) (3) (4)

1	$(\mathbb{1} \times \text{list}((R \times R)^\circ))\ (\mathbb{1} \times (\text{list}(g)\ \text{concat}))\ h$	$(R \times R)^\circ\ g \sqsubseteq g\ R^\circ$
2	$\sqsubseteq (\mathbb{1} \times (\text{list}(g\ R^\circ)\ \text{concat}))\ h$	$\text{list}(R^\circ)\ \text{concat} \sqsubseteq \text{concat}\ R^\circ$
3	$\sqsubseteq (\mathbb{1} \times (\text{list}(g)\ \text{concat}\ R^\circ))\ h$	$(\mathbb{1} \times R^\circ)\ h \sqsubseteq h\ R^\circ$
4	$\sqsubseteq (\mathbb{1} \times (\text{list}(g)\ \text{concat}))\ h\ R^\circ$	

g — $g := \pi_2$ is $(R \times R)^\circ \pi_2 = (\text{dom}(\pi_1 R^\circ))\pi_2 R^\circ \sqsubseteq \pi_2 R^\circ$, $g := \pi_1$ its mirror $(\text{dom}(\pi_2 R^\circ))\pi_1 R^\circ \sqsubseteq \pi_1 R^\circ$ — $\triangleq \times$ and $\langle R, S \rangle\ \pi_1 = \text{dom}(S)\ R$, $\langle R, S \rangle\ \pi_2 = \text{dom}(R)\ S$, then $\text{dom} \sqsubseteq \mathbb{1}$; $g := \text{choose} \triangleq \pi_1 \cup \pi_2$ is the union of the two $\text{lax}(\varphi) \wedge \text{lax}(\psi) \Longrightarrow G(R)\ (\varphi_B \cup \psi_B) \sqsubseteq (\varphi_A \cup \psi_A)\ F(R)$. list monotonic, $\triangleq$ relator

concat — a LEAF: no law above it. cost is a sum, so $\text{cost}(\text{concat}(xss)) = \text{sum}(\text{list}(\text{cost})(xss))$ and a cheaper part makes a cheaper whole.

h — a LEAF: $\text{cost}(\text{cons}(a, xs)) = \text{rating}(a) + \text{cost}(xs)$, so a cheaper tail makes a cheaper list. For $h := \pi_2$ it is an EQUALITY $(\mathbb{1} \times R^\circ)\pi_2 = (\text{dom}(\pi_1))\pi_2 R^\circ = \pi_2 R^\circ$: π_1 is a map, hence entire, so $\text{dom}(\pi_1) = \mathbb{1}$ — $\text{Entire}(R) \triangleq \text{dom}(R) = \mathbb{1}$

$(\mathbb{1} \times (\text{list}(g)\ \text{concat}))h$	g	h
include	π_2	cons
exclude	choose	π_2

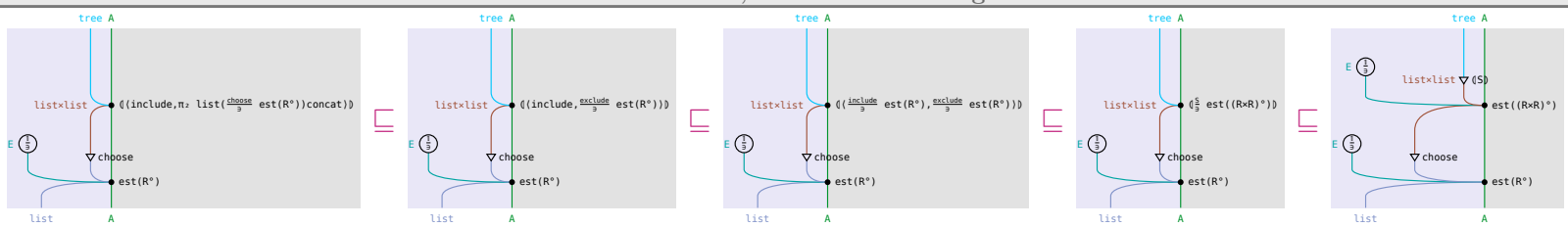
(7.3.3d)

1	$R \triangleq \{(0,0)\} : \{0,1\} \to \{0,1\}$	not entire — undefined at 1
2	$(R \times R)\text{choose}$ at $(0,1)$	nothing — $R \times R$ needs BOTH components related, and R is undefined at 1
3	choose R at $(0,1)$	0 — choose picks the first component 0, and R 0 0 holds
4	$((0,1),0)$	in choose R, not in $(R \times R)\text{choose}$
5	$(R \times R)\text{choose} = \text{choose}\ R$	iff R is entire — $\text{dom} \sqsubseteq \mathbb{1}$ is the g row's only inequality step, and it is an equality iff $\text{dom}(R) = \mathbb{1}$

§7.3.4. Derivation

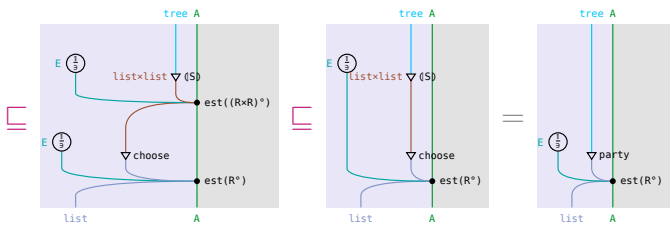
$$(\!|\text{include},\pi_2\ \text{list}(\tfrac{\text{choose}}{\ni}\ \text{est}(R^\circ))\ \text{concat}|\!)\ \tfrac{\text{choose}}{\ni}\ \text{est}(R^\circ) \sqsubseteq \tfrac{\text{party}}{\ni}\ \text{est}(R^\circ) \tag{7.3.4a}$$

the best of every guest list the president allows is one pass up the tree, each subtree handing up its best party with its boss in and its best with the boss out, and **choose** taking the better of the two at the root



(1) (2) (3) (4) (5)

1	$(\!\|\text{include},\pi_2\ \text{list}(\tfrac{\text{choose}}{\ni}\ \text{est}(R^\circ))\ \text{concat}\|\!)\ \tfrac{\text{choose}}{\ni}\ \text{est}(R^\circ)$	$\pi_2\ \text{list}(\tfrac{\text{choose}}{\ni}\ \text{est}(R^\circ))\ \text{concat} \sqsubseteq \tfrac{\text{exclude}}{\ni}\ \text{est}(R^\circ)$
2	$\sqsubseteq (\!\|\langle\text{include},\tfrac{\text{exclude}}{\ni}\ \text{est}(R^\circ)\rangle\|\!)\ \tfrac{\text{choose}}{\ni}\ \text{est}(R^\circ)$	$\text{include} \sqsubseteq \tfrac{\text{include}}{\ni}\ \text{est}(R^\circ)$
3	$\sqsubseteq (\!\|\langle\tfrac{\text{include}}{\ni}\ \text{est}(R^\circ),\tfrac{\text{exclude}}{\ni}\ \text{est}(R^\circ)\rangle\|\!)\ \tfrac{\text{choose}}{\ni}\ \text{est}(R^\circ)$	$\langle\tfrac{\text{include}}{\ni}\ \text{est}(R^\circ),\tfrac{\text{exclude}}{\ni}\ \text{est}(R^\circ)\rangle \sqsubseteq \tfrac{S}{\ni}\ \text{est}((R \times R)^\circ)$
4	$\sqsubseteq (\!\|\tfrac{S}{\ni}\ \text{est}((R \times R)^\circ)\|\!)\ \tfrac{\text{choose}}{\ni}\ \text{est}(R^\circ)$	$(\!\|\tfrac{S}{\ni}\ \text{est}((R \times R)^\circ)\|\!) \sqsubseteq \tfrac{(\!\|S\|\!)}{\ni}\ \text{est}((R \times R)^\circ)$
5	$\sqsubseteq \tfrac{(\!\|S\|\!)}{\ni}\ \text{est}((R \times R)^\circ)\ \tfrac{\text{choose}}{\ni}\ \text{est}(R^\circ)$	



(6) (7) (8)

6	$\sqsubseteq \tfrac{(\!\|S\|\!)}{\ni}\ \text{est}((R \times R)^\circ)\ \tfrac{\text{choose}}{\ni}\ \text{est}(R^\circ)$	$\text{est}((R \times R)^\circ)\ \tfrac{\text{choose}}{\ni}\ \text{est}(R^\circ) \sqsubseteq E(\text{choose})\ \text{est}(R^\circ)$ $\tfrac{(\!\|S\|\!)\ \text{choose}}{\ni} = \tfrac{(\!\|S\|\!)}{\ni}\ E(\text{choose})$
7	$\sqsubseteq \tfrac{(\!\|S\|\!)\ \text{choose}}{\ni}\ \text{est}(R^\circ)$	$\text{party} = (\!\|S\|\!)\ \text{choose}$
8	$= \tfrac{\text{party}}{\ni}\ \text{est}(R^\circ)$	

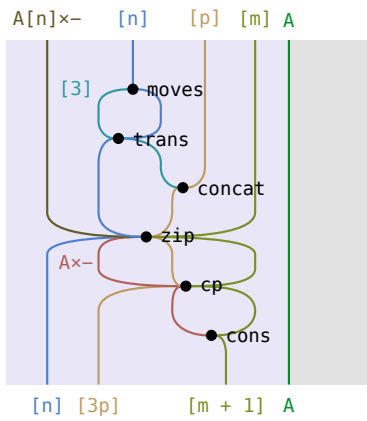
§7.4. Shortest paths on a cylinder

(7.4a)

		on L=list⁺, A.3.3	on Vec	note
1	F	F(A,X)=A+A×X	F(A,A[m])=A[1]+A[m+1]	A square alone, or in front of the path so far.
2	the input	L(N A)	A[m][n]≜(A[n])[m]	m columns of n squares, outermost index first — the matrix itself.
3	a path	L A, a non-empty list	A[1]≜A, A[m+1]≜A×A[m]	Through m columns, m squares: the count is in the type. A one-square path is the square; Vec(m) is the functor X↦X[m].
4	the paths into a row	P(L A)	A[p][m], p=3^(m-1)	Three moves per column crossed, so this count is in the type too; a set forgets it.
5	α	[wrap,cons] F(A,LA)→LA	wrap=𝟙:A→A[1] cons:A×A[m]→A[m+1]	wrap is an identity; cons is only an iso, so it stays a bead.
6	moves	NA→PNA	X[n]→X[3][n]	moves(x)=(up(x),x,down(x)), a 3×n matrix.
7	trans	PNX→NPX	X[3][n]→X[n][3]	The transpose.
8	union	PPX→PX	concat:X[j][k]→X[jk]	The j rows laid end to end; j is 3 inside gen and n inside paths.
9	zip	F(NA,NB)→NF(A,B)	F(A[n],B[n])→F(A,B)[n]	Unchanged.
10	cp	F(A,PB)→P(F(A,B))	A×B[p]→(A×B)[p]	The square paired with each of the p candidates.
11	est(R)≜∋∩(∈\R*)PX→X		X[k]→X, ∋:X[k]→X	A component that is R-below every component; ∋ is "is some component" and ∈ its converse.
12	R≜sum≤sum°	[ℕ]→[ℕ]	ℕ[m]→ℕ[m]	The cost of a path, which the cheapest minimises — one order per length m.
13	gen	F(𝟙,moves trans N(union)) zip N(cp P(α)) F(NA,N(P(LA))) →N(P(LA))	F(𝟙,moves trans Vec(n)(concat)) zip Vec(n)(cp Vec(3p)(cons)) F(A[n],A[n][p][m])→A[n][3p][m+1]	The same composite without α; the type shown is the cons side, the wrap side is 𝟙:A[n]→A[n].
14	⦇gen⦈	L(N A)→N(P(L A))	A[m][n]→A[n][p][m]	Columns in, rows out.
15	paths≜⦇gen⦈ concat	⦇gen⦈ setify union [A[n]]→P[A]	A[m][n]→A[np][m]	The n rows of p paths laid end to end: every path of the cylinder, in one row.
16	Q, 7.4.3	F(𝟙,moves trans N(est(R))) zip N(α)	F(𝟙,moves trans Vec(n)(est(R))) zip, F(A[n],A[n][m])→A[n][m+1]	The cheapest of three; no p anywhere. ⦇Q⦈:A[m][n]→A[n][m] has the type of a transpose.
17	the specification paths est(R)	[ℕ[n]]→[ℕ]	A[m][n]→A[m], est(R):X[np]→X	A cheapest path from the entry side to the exit side.

§7.4.1. gen ≜ (𝟙 × (moves trans Vec(concat))) zip Vec(cp Vec(cons)) — one more column: each square's paths go to its three neighbours, then the square goes in front

(7.4.1a)

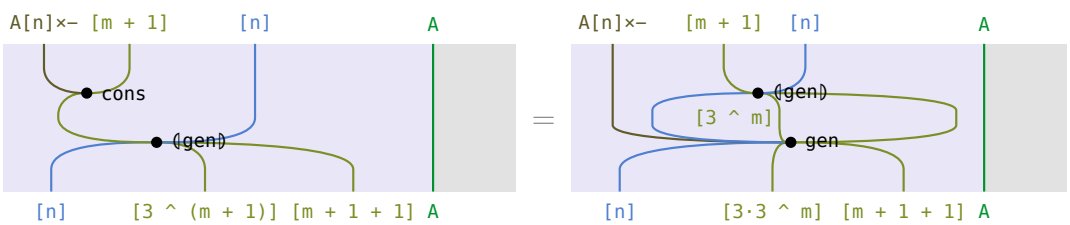


(7.4.1b)

```
a row is the outer index, a cell one path
u = ((1,2,3,4),C)                                  : A[n]×A[n]
    C = 5                                            one candidate per row, one square long:
        6                                            a column of squares
        7
        8
(𝟙×(moves trans Vec(n)(concat)))                     𝟙 keeps the column, the candidates move
  moves(C)                                           down, unmoved, up: a 3×n matrix
   = 6 7 8 5
     5 6 7 8
     8 5 6 7                                       : A[3][n]
  trans(that)                                        the transpose: row k gets rows k-1, k, k+1
   = 6 5 8
     7 6 5
     8 7 6
     5 8 7                                         : A[n][3]
  Vec(n)(concat)(that) = that                        concat:X[3][1]→X[3] is 𝟙
zip(that)                                            each row: its square, and its candidates
   = (1, 6 5 8)
     (2, 7 6 5)
     (3, 8 7 6)
     (4, 5 8 7)                                    : (A×A[3])[n]
  Vec(n)(cp)(that)                                   cp pairs the square with each candidate:
     = 1 6   2 7   3 8   4 5                         row k of that is the k-th 3×2 block, left
       1 5   2 6   3 7   4 8                         to right; each line a two-square path
       1 8   2 5   3 6   4 7                       : A[n][3][2]
```

§7.4.2. cons ⦇gen⦈ = (𝟙 × ⦇gen⦈) gen — folding a matrix with one more column is folding the rest, then one gen

(7.4.2a)



the `[m+1]` bracket puts the column back on the matrix, then the fold reads all of it

the fold reads the rest under `F`, then one `gen` puts the column in front

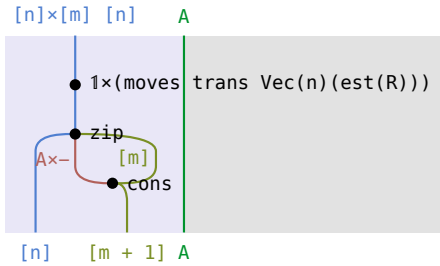
(7.4.2b)

```
xs = 1 2 3 4                                       : A[m][n], m=2
     5 6 7 8

⦇gen⦈(xs) = gen((1,2,3,4), ⦇gen⦈(5,6,7,8))
⦇gen⦈(5,6,7,8)                                       one column: each square its own one-square path
   = 5
     6
     7
     8                                             : A[n][1][1]
```

both halves are in `gen ≜ (𝟙 × (moves trans Vec(concat))) zip Vec(cp Vec(cons))` : the tail folds to the column `5 6 7 8`, one path per row and each of them one square long, and `gen` on it is that walk.

§7.4.3. Q ≜ (𝟙 × (moves trans Vec(n)(est(R)))) zip Vec(n)(cons) — Q is gen with est(R) choosing one path per square

(7.4.3a)



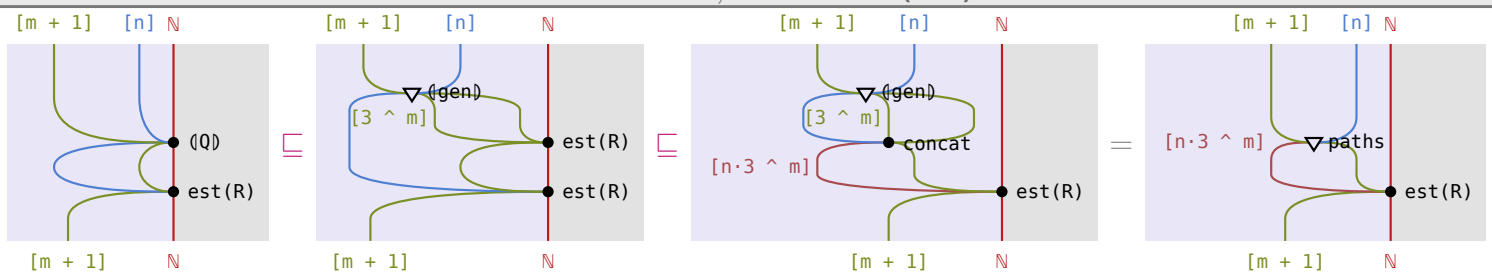
(7.4.3b)

```
u = ((1,2,3,4),C)                                  : A[n]×A[n][m]
    C = 5                                            one path per row, one square long
        6
        7
        8
(𝟙×(moves trans Vec(n)(est(R))))                     𝟙 keeps the column, the paths move
  moves(C)                                           down, unmoved, up: a 3×n matrix
   = 6 7 8 5
     5 6 7 8
     8 5 6 7                                       : A[3][n][m]
  trans(that)                                        the transpose: row k gets rows k-1, k, k+1
   = 6 5 8
     7 6 5
     8 7 6
     5 8 7                                         : A[n][3][m]
  Vec(n)(est(R))(that)                               the cheapest into each row – chosen BEFORE
   = 5 5 6 5                                       : A[n][m]   the new square is put in front
zip(that)                                            each row: its square, and the one predecessor
   = 1 5                                             that survived; the `[m+1]` bracket is cons
     2 5
     3 6
     4 5                                           : A[n][m+1]
```

§7.4.4. ⦇Q⦈ est(R) ⊑ paths est(R) — folding Q gives a smallest of paths

⦇Q⦈ est(R) ⊑ paths est(R) (7.4.4a)

a cheapest of all **np** paths of the cylinder is beaten by the greedy fold's one path per row and then a cheapest of those **n**, which costs **O(n×m)**.



(1) (2) (3) (4)

1	⦇Q⦈ est(R)	⦇Q⦈ ⊑ ⦇gen⦈ Vec(n)(est(R))
2	⊑ ⦇gen⦈ Vec(n)(est(R)) est(R)	R R ⊑ R
3	⊑ ⦇gen⦈ concat est(R)	f g = f g
4	= paths est(R)	

S S ⊑ S ⟹ Vec(j)(est(S)) est(S) ⊑ concat est(S) , R transitive

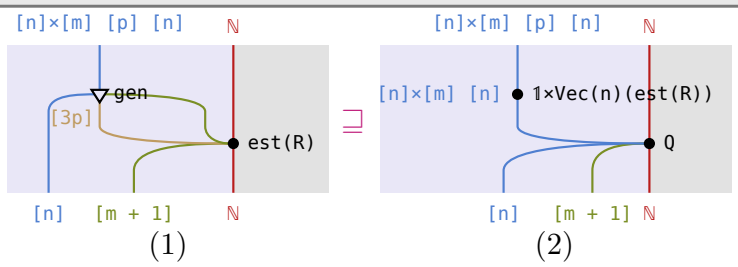
a cheapest of each of the **j** rows and then a cheapest of those **j** is a cheapest of all **jk** entries laid end to end.

⦇Q⦈ ⊑ ⦇gen⦈ Vec(n)(est(R))

the one path per row the greedy fold keeps is one of the **p** that ⦇gen⦈ generates for that row, and a cheapest of them.

(𝟙 × Vec(n)(est(R))) Q ⊑ gen Vec(n)(est(R)) (7.13)

choosing a cheapest of each square's **p** paths before the column is extended is no better than extending first and choosing among the **3p**.



(1) (2)

1	gen Vec(n)(est(R))	(7.13); zip, trans, moves lax natural
2	⊑ (𝟙 × Vec(n)(est(R))) Q	

(7.13) on Vec: (𝟙 × est(R)) cons ⊑ cp Vec(k)(cons) est(R)

putting the new square in front of every one of the **p** candidates and then choosing a cheapest is beaten by choosing a cheapest first and putting the square in front of that one.

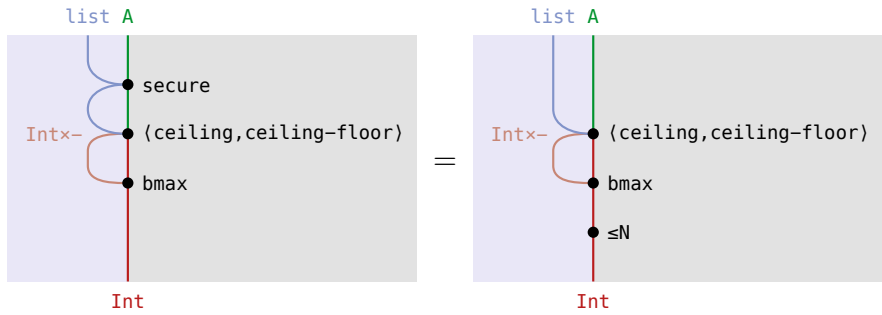
§7.5. The security van problem

(7.5a)

	definition	type	example
1	R≜length≤length°	[[A]]→[[A]]	[[a,b,c]] to [[a],[b,c]], and not back: 1≤2.
2	ceiling≜ $\frac{\text{prefix sum}}{\ni}$ est(≥)	[A]→Int	ceiling[a,b] is the largest of 0, a, a+b.
3	floor≜ $\frac{\text{prefix sum}}{\ni}$ est(≤)	[A]→Int	floor[a,b] is the smallest of 0, a, a+b, so [a,b] carries ceiling−floor cash.
4	secure the coreflexive on x with bmax(ceiling x,ceiling x−floor x)≤N	[A]→[A]	secure keeps [a] exactly when −N≤a≤N.
5	ok the coreflexive on (a,xs) with xs non-empty and [a]+head(xs) secure	A×[[A]]→A×[[A]]	ok keeps (a,[[b],[c]]) exactly when [a,b] is secure.
6	new≜(wrap×𝟙) cons	A×[[A]]→[[A]]	new(a,[[b],[c]])=[[a],[b],[c]].
7	glue≜(𝟙×cons°) assocl (cons×𝟙) cons	A×[[A]]→[[A]]	glue(a,[[b],[c]])=[[a,b],[c]].
8	old≜(𝟙×cons°) assocl ((cons secure)×𝟙) cons	A×[[A]]→[[A]]	old(a,[[b],[c]])=[[a,b],[c]] when [a,b] is secure, and nothing otherwise.
9	partition = ⦇[nil,new ∪ glue]⦈ cutting the transactions into non-empty segments is one pass along them that either opens a segment for the transaction in hand or puts it on the front of the segment already open	[A]→[[A]]	partition[a,b] gives [[a],[b]] and [[a,b]].
10	S≜[nil,new ∪ old]	F[[A]]→[[A]]	S(a,[[b],[c]]) gives [[a],[b],[c]], and [[a,b],[c]] when [a,b] is secure.
11	(∀a. secure(N,[a])) ⟹ partition list(secure) = ⦇S⦈ cutting the transactions into non-empty segments every way and keeping the cuts whose every segment is secure is the one fold whose algebra is [nil,new ∪ old]	[A]→[[A]]	Both take [a,b] to [[a],[b]] — each [a] is secure — and to [[a,b]] when [a,b] is.
12	H≜(head prefix° head°) ∪ (nil° nil)	[[A]]→[[A]]	[[a],[b,c]] to [[a,b],[c]]: [a] is a prefix of [a,b].
13	R;H≜R∩(R°⇒H)	[[A]]→[[A]]	[[a,b,c]] to [[a],[b,c]] by |R|, and [[a],[b,c]] to [[a,b],[c]] by R∩H.
14	R;H = |R| ∪ R ∩ H |R| is the strict part of R, strictly fewer segments; R;H splits into it and R∩H	[[A]]→[[A]]	[[a,b,c]] to [[a],[b,c]] by |R|: 1<2.
15	the specification $\frac{\text{partition list(secure)}}{\ni}$ est(R)	[A]→[[A]]	[a,b] gives [[a,b]] when [a,b] is secure, and [[a],[b]] otherwise.

§7.5.1. secure prefix ⊑ prefix secure — a prefix of a secure segment is secure

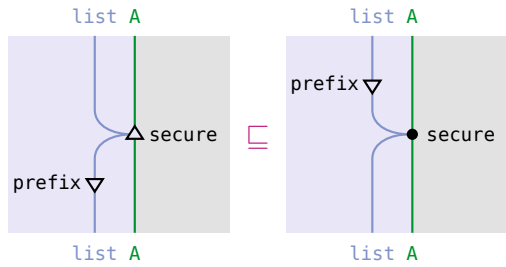
(7.5.1a)



`secure ⟨ceiling,ceiling−floor⟩ bmax = ⟨ceiling,ceiling−floor⟩ bmax (≤N)`

a stretch passes `secure` before the test exactly where the test's own value passes `≤N` after it, which is `secure(N,xs)` ≜ `bmax(ceiling(xs),ceiling(xs)−floor(xs)) ≤ N` 's `bmax(ceiling x,ceiling x−floor x)≤N`

Exercise 7.49

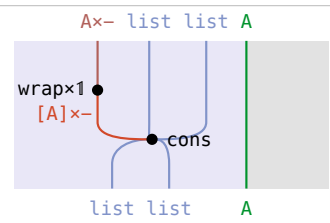
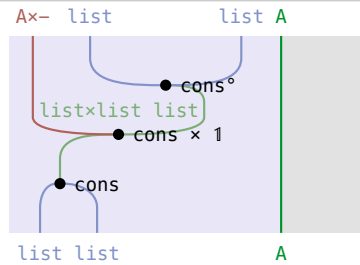
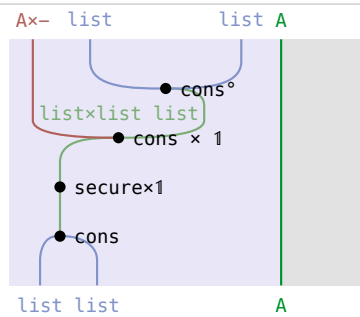


`secure prefix ⊑ prefix secure`

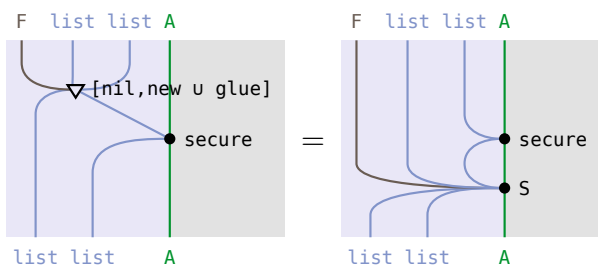
every pair `secure` then `prefix` gives, `prefix` then `secure` gives too — a prefix of a secure stretch is itself secure, which is the prefix-closure B&dM p.185 names

§7.5.2. `(∀a. secure(N,[a])) ⟹ partition list(secure) = ⦇S⦈` — partitioning into secure segments is one fold

(7.5.2a)

	definition	Hinze–Marsden
1	`new≜(wrap×𝟙) cons` `wrap` makes the transaction a segment on its own, and `cons` puts that segment at the front of the schedule: the van is called	
2	`glue≜(𝟙×cons°) assocl (cons×𝟙) cons` `cons°` splits the schedule into its first segment and the rest, the first `cons` puts the transaction at the front of that segment, and the second puts the segment back	
3	`old≜(𝟙×cons°) assocl ((cons secure)×𝟙) cons` `glue` with `secure` on the segment the transaction has just joined: the van stays away only where the longer segment still passes	

(7.5.2b)

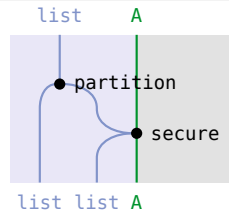
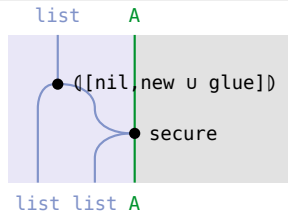
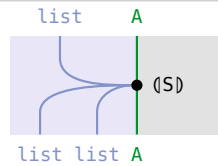


`(∀a. secure(N,[a])) ⟹ [nil,new ∪ glue] list(secure) = F(list(secure)) S`

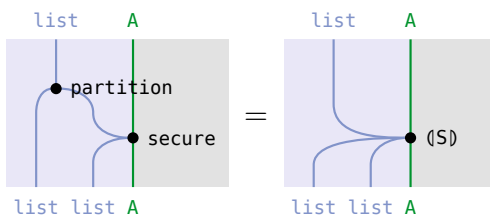
testing every segment for security after the algebra has run is testing it before it ran: a segment `glue` has extended is secure exactly where `old` would have let it be, and the one transaction `new` sets out on its own is secure whatever it is

(∀a. secure(N,[a])) ⟹ partition list(secure) = ⦇S⦈ Exercise 7.51

cutting the transactions into non-empty segments every way and keeping the cuts whose every segment is secure is the one fold whose algebra is `[nil,new ∪ old]`

step	Hinze–Marsden
`partition list(secure)` the specification: cut the transactions every way, then keep the cuts whose every segment is secure	
= `⦇[nil,new ∪ glue]⦈ list(secure)` `partition` is the fold that either opens a segment for the transaction in hand or puts it on the front of the segment already open	
= `⦇[nil,new ∪ old]⦈` fusion: the condition above moves `list(secure)` inside that fold, which is what turns `glue` into `old`	

(7.5.2c)



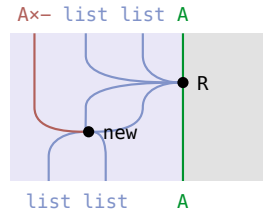
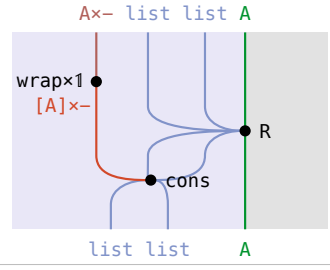
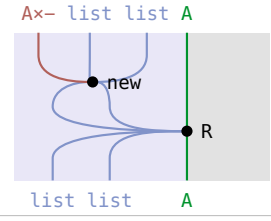
`(∀a. secure(N,[a])) ⟹ partition list(secure) = ⦇S⦈`

cutting the transactions every way and then keeping the cuts whose every segment is secure is one pass along them that either calls the van or extends the open segment while it stays secure

§7.5.3. `(𝟙 × R) new ⊑ (new ∪ old) R` — new on an R-related schedule is matched by new or old

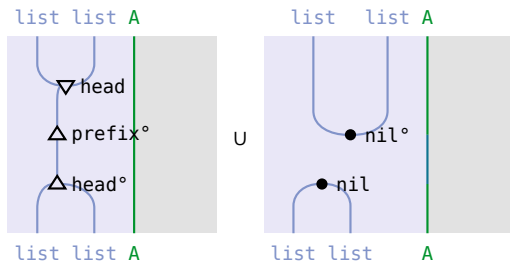
(𝟙 × R) new ⊑ (new ∪ old) R (7.14)

calling the van for the transaction on a no-longer schedule gets no further than calling it on this one and shortening the schedule afterwards

step	Hinze–Marsden
`(𝟙×R)new` the left side of (7.14) — `R = clen cast ≤ clen cast°` , `new(p) ≜ [[π₁(p)]]#π₂(p)`	
= `(wrap×R)cons` `new≜(wrap×𝟙)cons` — `(wrap × 𝟙) cons` — and `(wrap×𝟙)(𝟙×R)=(wrap×R)`	
⊑ `new R` `cons` is monotonic on `R` — `(𝟙×R)cons⊑cons R`, consing onto a no-longer schedule leaves it no longer — and `new⊑new ∪ old`	

§7.5.4. `H ≜ (head prefix° head°) ∪ (nil° nil)` — the right first segment is a prefix of the left one, or both are empty

(7.5.4a)

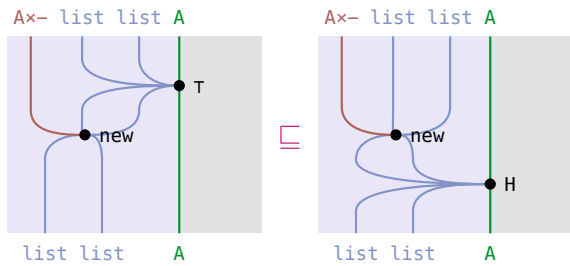


`H ≜ (head prefix° head°) ∪ (nil° nil)`

one schedule's first segment is a prefix of the other's, or both schedules are empty

§7.5.5. `(𝟙 × (R;H)) new ⊑ (new ∪ old) (R;H)` — `new` on an `R;H`-related schedule is matched by `new` or `old`

(7.18)

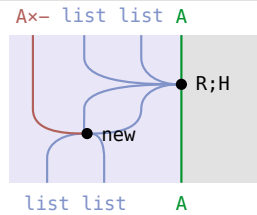
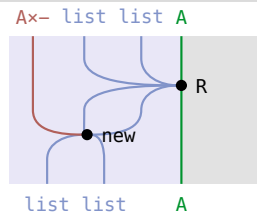
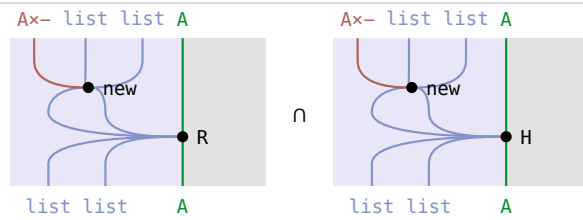
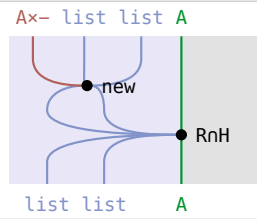
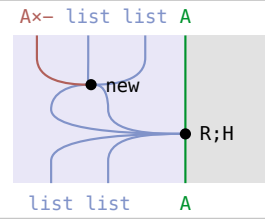


`(𝟙 × T) new ⊑ new H`

whatever schedule the van is called on, the result's first segment is the one transaction `[a]`, so any two results of `new` on that transaction stand in `H`

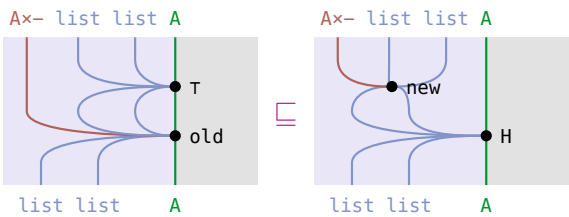
`(𝟙 × (R;H)) new ⊑ (new ∪ old) (R;H)` (7.16)

calling the van for the transaction on a `R;H`-better schedule gets no further than calling it on this one and bettering the whole schedule afterwards

step	Hinze–Marsden
`(𝟙×(R;H))new` the left side of (7.16)	
`⊑ (𝟙×R)new` `R;H⊑R` — `R;H = R ∩ (R° ⇨ H)`	
`⊑ new R ∩ new H` (7.14) as far as its `new R` line — (7.14) `(𝟙 × R) new ⊑ (new ∪ old) R` — and (7.18)	
`= new (R∩H)` `new` is a map, and a map distributes over `∩`	
`⊑ new (R;H)` `X∩Y⊑X;Y`, and `new⊑new ∪ old`	

§7.5.6. `(𝟙 × (R;H)) old ⊑ (new ∪ old) (R;H)` — `old` on an `R;H`-related schedule is matched by `new` or `old`

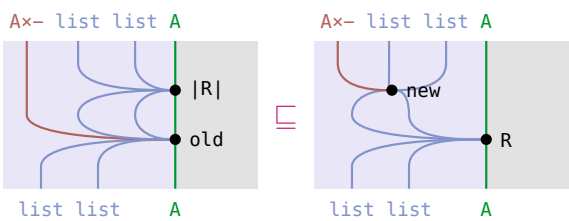
(7.19)



`(𝟙 × T) old ⊑ new H`

`old` leaves the transaction `[a]` at the front of the first segment, and `[a]` is what `new` makes that segment, so the two first segments are `prefix`-related whatever the schedules were

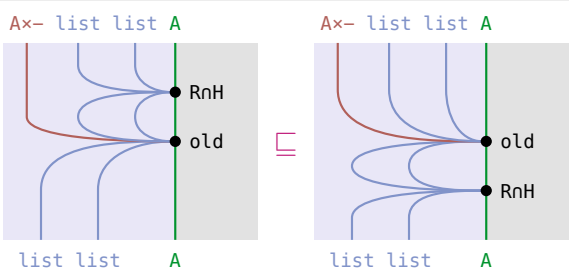
(7.20)



`(𝟙 × |R|) old ⊑ new R`

`old` keeps the schedule's length, so a strictly shorter one still comes out no longer than the one the van's own segment lengthens

(7.21)

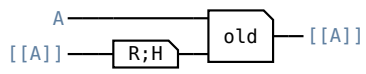
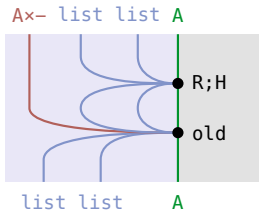
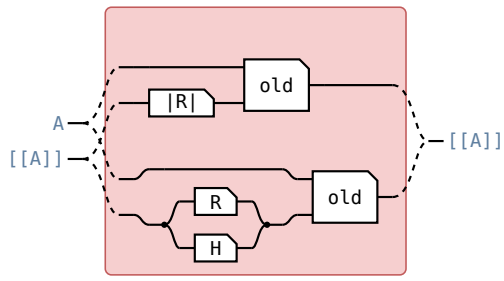
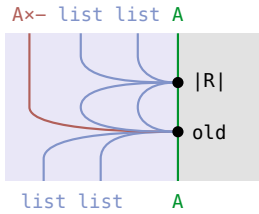
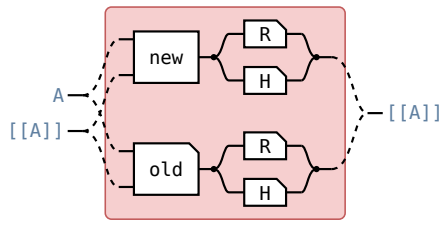
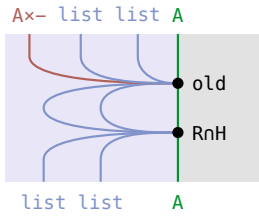
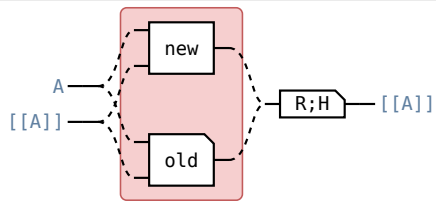
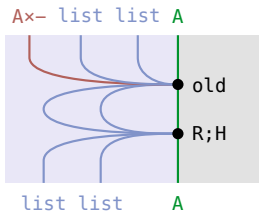


`(𝟙 × (R ∩ H)) old ⊑ old (R ∩ H)`

on a tie the one first segment is a prefix of the other, so prefix-closure of `secure` — Exercise 7.49 `secure prefix ⊑ prefix secure` — lets `old` fire on this side too, and it keeps both the length and the prefix

(𝟙 × (R;H)) old ⊑ (new ∪ old) (R;H) (7.17)

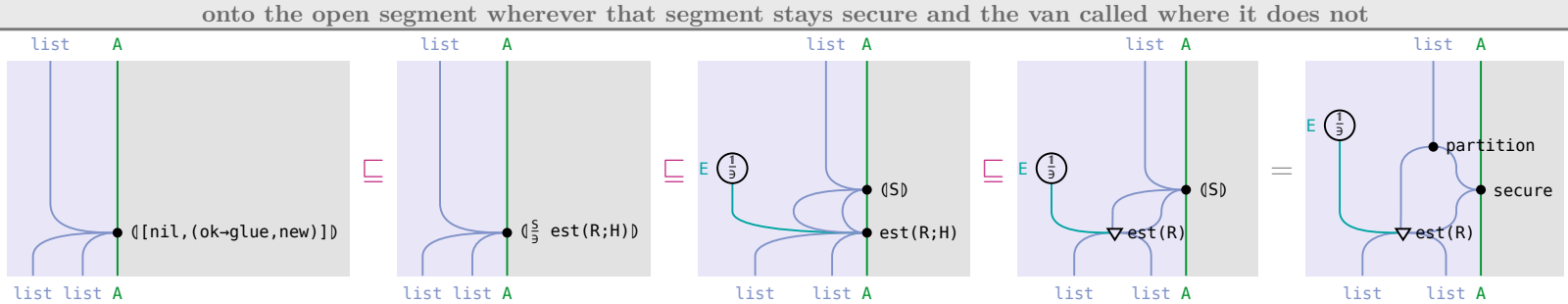
gluing the transaction onto a better schedule for the rest gets no further than gluing it on, or calling the van, and bettering the whole schedule after,

circuit — the `old` branch of each union	**Hinze–Marsden**
	 the `old` operand of `new ∪ old`, in every row
=  `((𝟙 × |R|) old) ∪ ((𝟙 × (R ∩ H)) old)` `R;H=|R| ∪ (R∩H)` — `R;H = |R| ∪ R ∩ H`, ∪ distributes,	
⊑  `(new (R ∩ H)) ∪ (old (R ∩ H))` (7.19) and (7.20) on `|R|` — (7.19) `(𝟙 × ⊤) old ⊑ new H`, (7.20) `(𝟙 × |R|) old ⊑ new R` — and (7.21) on `R∩H` — (7.21) `(𝟙 × (R ∩ H)) old ⊑ old (R ∩ H)`	
⊑  `(new ∪ old) (R;H)` `X∩Y⊑X;Y`, converses	

§7.5.7. Derivation

(∀a. secure(N,[a])) ⟹ ⦇[nil,(ok→glue,new)]⦈ ⊑ $\frac{\text{partition list(secure)}}{\ni}$ est(R) (7.5.7a)

the fewest secure segments the transactions can be cut into are one pass along them, the next transaction glued onto the open segment wherever that segment stays secure and the van called where it does not



(1) (2) (3) (4) (5)

1	⦇[nil,(ok→glue,new)]⦈	`[nil,(ok→glue,new)] ⊑ S/∋ est(R;H)`
2	⊑ ⦇S/∋ est(R;H)⦈	⦇S/∋ est(R;H)⦈ ⊑ ⦇S⦈/∋ est(R;H)
3	⊑ ⦇S⦈/∋ est(R;H)	`R;H ⊑ R`
4	⊑ ⦇S⦈/∋ est(R)	Exercise 7.51 (∀a. secure(N,[a])) ⟹ partition list(secure) = ⦇S⦈
5	= $\frac{\text{partition list(secure)}}{\ni}$ est(R)	

§8. Thinning Algorithms

§8.1. Thinning

(8.1)

`thin(Q) ≜ ⊇ ∩ (∈ \ (Q° ∈))` — `thin(Q)` takes a set `ys` to a subset `xs` that keeps a `Q`-lower bound of every `a∈ys`.

`thin(≤) ys thin(≤) xs⟺xs⊆ys∧(∀a∈ys. ∃b∈xs. b≤a)`.

(8.1a)

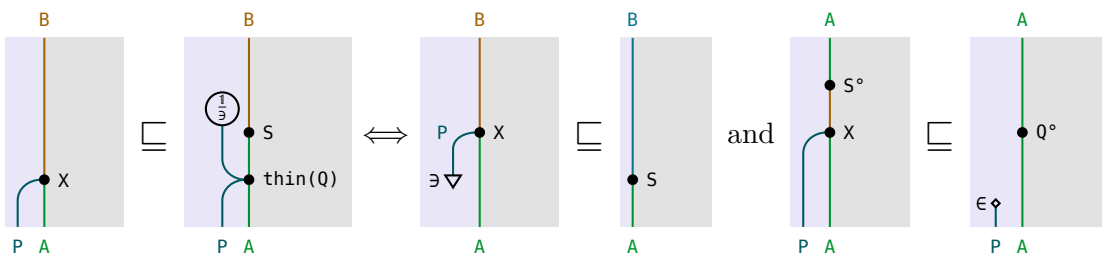
	the law	what it says
1	`X ⊑ S/∋ thin(Q) ⟺ X ∋ ⊑ S ∧ S° X ⊑ Q° ∈`	`S` is the whole algebra: it produces every candidate that `thin(Q)` thins. Everything kept is an `S`-value, and every `S`-value has a `Q`-lower bound among the kept ones
2	`thin(Q) ∋ ⊑ ∋`	everything a thinning keeps was in the set
3	`thin(Q)° ∋ ⊑ ∋ Q`	every element of the set has a `Q`-lower bound among the kept ones
4	`Q ⊑ R ⟹ thin(Q) ⊑ thin(R)`	the fewer pairs `Q` relates, the fewer subsets count as thinnings
5	`𝟙 ⊑ Q ⟹ 𝟙 ⊑ thin(Q)` `Q Q ⊑ Q ⟹ thin(Q) thin(Q) ⊑ thin(Q)`	keeping everything is always a legal thinning

6	`Q ⊑ R ∧ 𝟙 ⊑ Q ∧ R R ⊑ R ⟹ est(R) = thin(Q) est(R)`
	`𝟙 ⊑ Q`
	`⟹ est(R) ⊑ thin(Q) est(R)`
		thin-introduction: thinning first cannot lose an `R`-minimum

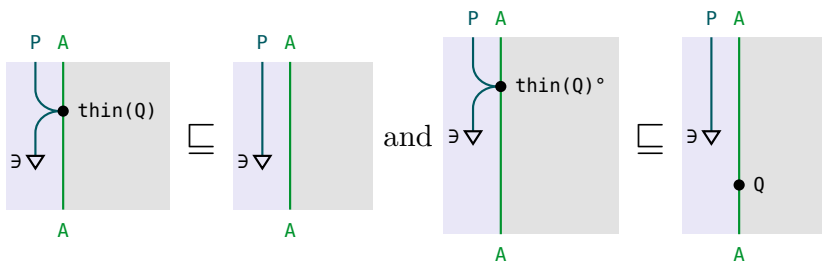
7	`thin(Q)⊒est(Q) 𝟙/∋` (8.2) — (8.2) `est(Q) 𝟙/∋ ⊑ thin(Q)`	**thin-elimination**: keeping one element is a thinning, but its domain is the sets `est(Q)` is defined on
8	`S/∋ thin(Q)⊒ S/∋ est(R) 𝟙/∋` (8.3), `R∩(S°S)⊑Q` — thin-elimination variant	the usable variant: `R` need only refine `Q` between values `S` gives one argument
9	`P(thin(Q)) union ⊑ union thin(Q)` (8.4) — (8.4) `P(thin(Q)) union ⊑ union thin(Q)`	thinning each member set is a thinning of the union

§8.1.1. `X ⊑ S/∋ thin(Q) ⟺ X ∋ ⊑ S ∧ S° X ⊑ Q° ∈` — the definition of `S/∋ thin(Q)`

(8.1.1a)



(8.1.1b)

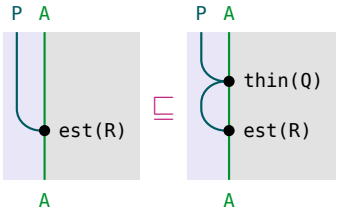


`X ⊑ S/∋ thin(Q) ⟺ X ∋ ⊑ S ∧ S° X ⊑ Q° ∈` at `X≜thin(Q)`, `S≜∋`: the left side is `thin(Q)⊑thin(Q)`, since `∋/∋=Λ(∋)=𝟙`

§8.1.2. `Q ⊑ R ∧ 𝟙 ⊑ Q ∧ R R ⊑ R ⟹ est(R) = thin(Q) est(R)` — thinning first, then taking the smallest, is the same as taking the smallest

$\mathbb{1} \sqsubseteq Q \implies \mathtt{est}(R) \sqsubseteq \mathtt{thin}(Q)\ \mathtt{est}(R)$ (8.1.2a)

the $\sqsubseteq$ half: keeping everything is a thinning — $\mathbb{1} \sqsubseteq Q$

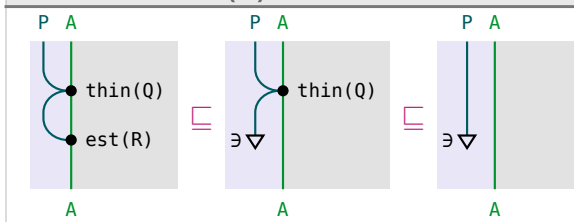


(1) (2)

1	est(R)	$\mathbb{1} \sqsubseteq \mathtt{thin}(Q)$ — $\mathbb{1} \sqsubseteq Q$ $\implies \mathbb{1} \sqsubseteq \mathtt{thin}(Q)$, Q $Q \sqsubseteq Q$ $\implies \mathtt{thin}(Q)\ \mathtt{thin}(Q) \sqsubseteq \mathtt{thin}(Q)$
2	$\sqsubseteq$ thin(Q) est(R)	

$\mathtt{thin}(Q)\ \mathtt{est}(R) \sqsubseteq \ni$ (8.1.2b)

the $\sqsupseteq$ half, first condition of the UP of est at $X \triangleq \mathtt{thin}(Q)\ \mathtt{est}(R)$ — $X \sqsubseteq \mathtt{est}(R) \iff X \sqsubseteq \ni \wedge \in X \sqsubseteq R^\circ$

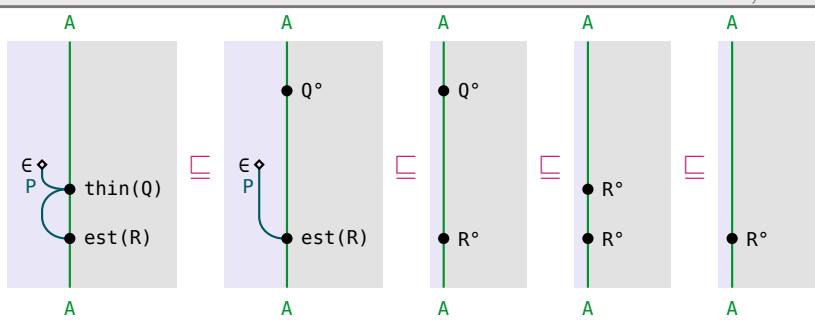


(1) (2) (3)

1	thin(Q) est(R)	$R \cap S \sqsubseteq R$
2	$\sqsubseteq$ thin(Q) $\ni$	thin(Q) $\ni \sqsubseteq \ni$
3	$\sqsubseteq \ni$	

$Q \sqsubseteq R \wedge R\ R \sqsubseteq R \implies \in \mathtt{thin}(Q)\ \mathtt{est}(R) \sqsubseteq R^\circ$ (8.1.2c)

the $\sqsupseteq$ half, second condition



(1) (2) (3) (4) (5)

1	$\in$ thin(Q) est(R)	$\in \mathtt{thin}(Q) \sqsubseteq Q^\circ \in$
2	$\sqsubseteq Q^\circ \in \mathtt{est}(R)$	$\in \mathtt{est}(R) \sqsubseteq R^\circ$
3	$\sqsubseteq Q^\circ\ R^\circ$	$Q \sqsubseteq R$
4	$\sqsubseteq R^\circ\ R^\circ$	$R\ R \sqsubseteq R$ $\implies R^\circ\ R^\circ \sqsubseteq R^\circ$
5	$\sqsubseteq R^\circ$	

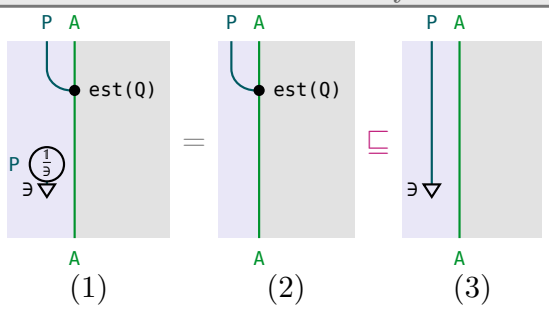
§8.1.3. Properties of thinning

est(Q) $\frac{1}{\ni}$ ⊑ thin(Q) (8.2)

the singleton holding a **Q**-least member of a set is a thinning of that set

est(Q) $\frac{1}{\ni}$ ∋ ⊑ ∋

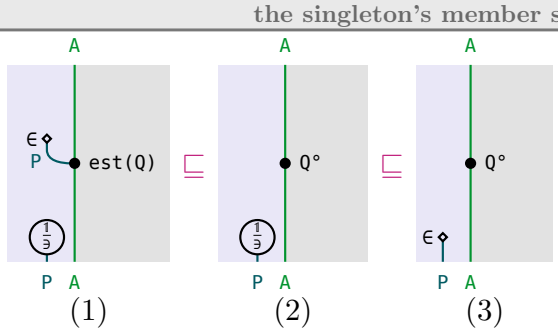
every member of the singleton is a member of the set — ≜ **thin**, ∋/∋ half



1	est(Q) $\frac{1}{\ni}$ ∋	$\frac{1}{\ni}$ ∋ = 𝟙
2	= est(Q)	R ∩ S ⊑ R
3	⊑ ∋	

∈ est(Q) $\frac{1}{\ni}$ ⊑ Q° ∈

the singleton's member stands in **Q** to every member of the set — ≜ **thin**, ∈\(Q°∈) half

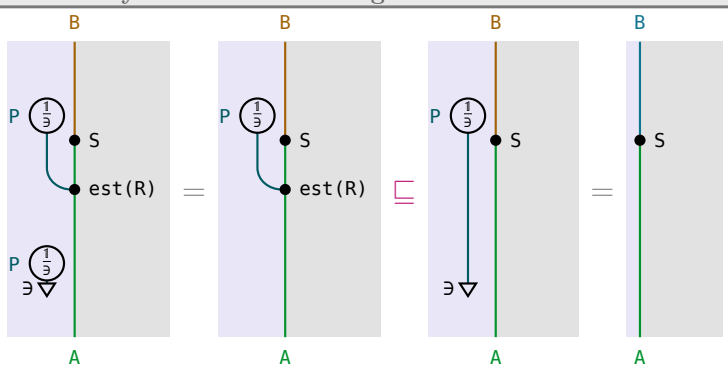


1	∈ est(Q) $\frac{1}{\ni}$	∈ est(R) ⊑ R°
2	⊑ Q° $\frac{1}{\ni}$	$\frac{1}{\ni}$ ⊑ ∈
3	⊑ Q° ∈	

$$\mathtt{R} \cap (\mathtt{S}^\circ\ \mathtt{S}) \sqsubseteq \mathtt{Q} \implies \tfrac{\mathtt{S}}{\ni}\ \mathtt{est(R)}\ \tfrac{\mathbb{1}}{\ni} \sqsubseteq \tfrac{\mathtt{S}}{\ni}\ \mathtt{thin(Q)} \tag{8.3}$$

$$\tfrac{\mathtt{S}}{\ni}\ \mathtt{est(R)}\ \tfrac{\mathbb{1}}{\ni}\ \ni \sqsubseteq \mathtt{S}$$

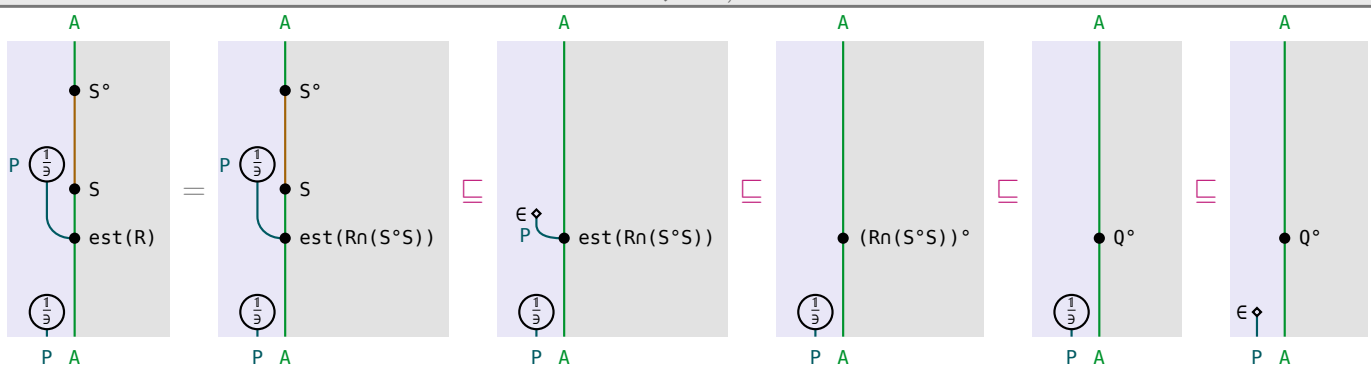
every member of the singleton is a value $\mathtt{S}$ returns — $\mathtt{X} \sqsubseteq \frac{\mathtt{S}}{\ni}\ \mathtt{thin(Q)} \iff \mathtt{X}\ \ni \sqsubseteq \mathtt{S} \wedge \mathtt{S}^\circ\ \mathtt{X} \sqsubseteq \mathtt{Q}^\circ\ \in$, first condition



1	$\frac{\mathtt{S}}{\ni}\ \mathtt{est(R)}\ \frac{\mathbb{1}}{\ni}\ \ni$	$\frac{\mathbb{1}}{\ni}\ \ni = \mathbb{1}$
2	$= \frac{\mathtt{S}}{\ni}\ \mathtt{est(R)}$	$\mathtt{R} \cap \mathtt{S} \sqsubseteq \mathtt{R}$
3	$\sqsubseteq \frac{\mathtt{S}}{\ni}\ \ni$	$\frac{\mathtt{R}}{\ni}\ \ni = \mathtt{R}$
4	$= \mathtt{S}$	

$$\mathtt{R} \cap (\mathtt{S}^\circ\ \mathtt{S}) \sqsubseteq \mathtt{Q} \implies \mathtt{S}^\circ\ \tfrac{\mathtt{S}}{\ni}\ \mathtt{est(R)}\ \tfrac{\mathbb{1}}{\ni} \sqsubseteq \mathtt{Q}^\circ\ \in$$

the singleton's member stands in $\mathtt{Q}$ to every value $\mathtt{S}$ returns at the same argument — $\mathtt{X} \sqsubseteq \frac{\mathtt{S}}{\ni}\ \mathtt{thin(Q)} \iff \mathtt{X}\ \ni \sqsubseteq \mathtt{S} \wedge \mathtt{S}^\circ\ \mathtt{X} \sqsubseteq \mathtt{Q}^\circ\ \in$, second condition



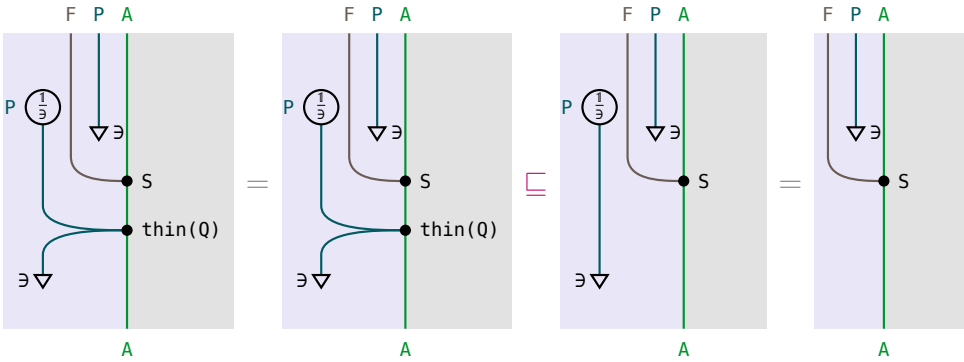
1	$\mathtt{S}^\circ\ \frac{\mathtt{S}}{\ni}\ \mathtt{est(R)}\ \frac{\mathbb{1}}{\ni}$	$\frac{\mathtt{S}}{\ni}\ \mathtt{est(R} \cap (\mathtt{S}^\circ\ \mathtt{S})) = \frac{\mathtt{S}}{\ni}\ \mathtt{est(R)}$
2	$= \mathtt{S}^\circ\ \frac{\mathtt{S}}{\ni}\ \mathtt{est(R} \cap (\mathtt{S}^\circ\ \mathtt{S}))\ \frac{\mathbb{1}}{\ni}$	$\mathtt{T}^\circ\ \frac{\mathtt{T}}{\ni} \sqsubseteq \in$
3	$\sqsubseteq \in\ \mathtt{est(R} \cap (\mathtt{S}^\circ\ \mathtt{S}))\ \frac{\mathbb{1}}{\ni}$	$\in\ \mathtt{est(R)} \sqsubseteq \mathtt{R}^\circ$
4	$\sqsubseteq (\mathtt{R} \cap (\mathtt{S}^\circ\ \mathtt{S}))^\circ\ \frac{\mathbb{1}}{\ni}$	$\mathtt{R} \cap (\mathtt{S}^\circ\ \mathtt{S}) \sqsubseteq \mathtt{Q}$
5	$\sqsubseteq \mathtt{Q}^\circ\ \frac{\mathbb{1}}{\ni}$	$\frac{\mathbb{1}}{\ni} \sqsubseteq \in$
6	$\sqsubseteq \mathtt{Q}^\circ\ \in$	

preorder(Q) ∧ S Pres Q ⟹ ⦇F(∋) S / ∋ thin(Q)⦈ ⊑ ⦇S⦈ / ∋ thin(Q) — Theorem 8.1

thinning at every step of the reduce is a thinning of the whole candidate set

⦇F(∋) S / ∋ thin(Q)⦈ ∋ ⊑ ⦇S⦈

everything the thinning fold keeps is a value of the plain fold — **X ⊑ S / ∋ thin(Q) ⟺ X ∋ ⊑ S ∧ S° X ⊑ Q° ∈**, first condition, by fusion from the chain below

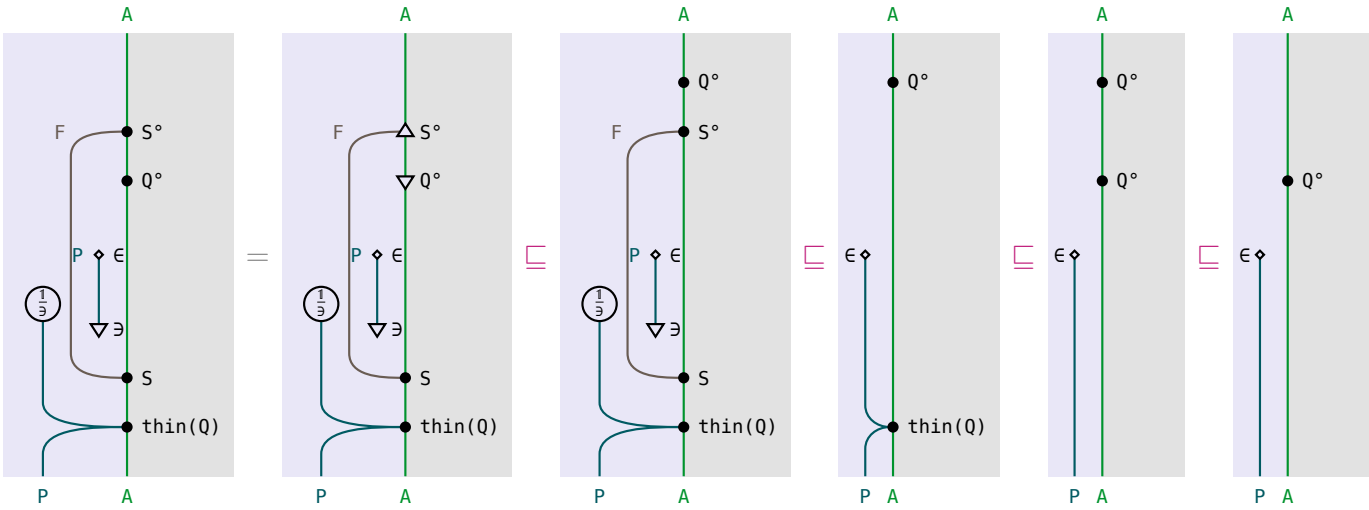


(1) (2) (3) (4)

1	F(∋) S / ∋ thin(Q) ∋	
2	= F(∋) S / ∋ thin(Q) ∋	thin(Q) ∋ ⊑ ∋
3	⊑ F(∋) S / ∋ ∋	R / ∋ ∋ = R
4	= F(∋) S	

preorder(Q) ∧ S Pres Q ⟹ ⦇S⦈° ⦇F(∋) S / ∋ thin(Q)⦈ ⊑ Q° ∈

every value of the plain fold is **Q**-above one the thinning fold keeps — **X ⊑ S / ∋ thin(Q) ⟺ X ∋ ⊑ S ∧ S° X ⊑ Q° ∈**, second condition, by hylomorphism theorem from the chain below



(1) (2) (3) (4) (5) (6)

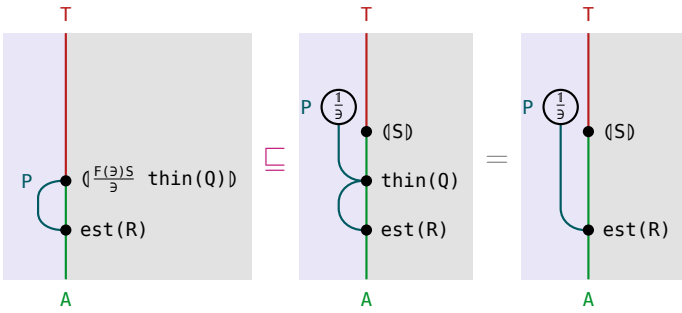
1	S° F(Q° ∈) F(∋) S / ∋ thin(Q)	
2	= S° F(Q°) F(∈) F(∋) S / ∋ thin(Q)	S Pres Q ⟹ S° F(Q°) ⊑ Q° S°
3	⊑ Q° S° F(∈) F(∋) S / ∋ thin(Q)	S° F(∈) F(∋) S / ∋ ⊑ ∈
4	⊑ Q° ∈ thin(Q)	∈ thin(Q) ⊑ Q° ∈
5	⊑ Q° Q° ∈	
6	⊑ Q° ∈	

(8.1.3a)

	step	type	example	what it does
1	S:F(A)→A is 8.1′s algebra; example F(X)=V+V×X, S=[wrap,cons], candidate a=cons(v,p)			
2	S°	A→FA	cons(v,p) ↦ r(v,p)	takes a apart into the input of its last step
3	F(Q°∈)	FA→F(PA)	r(v,p) ↦ r(v,ps), some p'∈ps with p' Q p	replaces each recursive component by a set holding a Q-no-worse one; v stays
4	F(∋)S / ∋	F(PA)→PA	r(v,ps) ↦ {cons(v,t) | t∈ps}	picks one element from each set, combines by S, collects every result
5	thin(Q)	PA→PA	{cons(v,t) | t∈ps} ↦ ys	keeps a subset; every dropped element is Q-beaten by a kept one
6	⊑ Q°∈	A→PA	ys holds y with y Q cons(v,p') Q cons(v,p) = a	the thinned set still holds an element Q-no-worse than a (S monotonic on Q, Q transitive)

$Q \sqsubseteq R \land \mathrm{preorder}(Q) \land \mathrm{preorder}(R) \land S \ \mathrm{Pres}\ Q \implies (\!|\frac{F(\ni)\ S}{\ni}\ \mathrm{thin}(Q)|\!)\ \mathrm{est}(R) \sqsubseteq \frac{(\!|S|\!)}{\ni}\ \mathrm{est}(R)$ Corollary 8.1

the thinning fold refines the optimisation problem itself



(1) (2) (3)

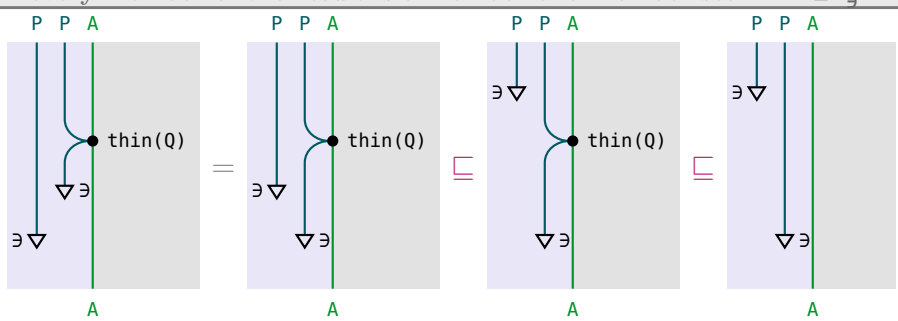
1	$(\!\|\frac{F(\ni)\ S}{\ni}\ \mathrm{thin}(Q)\|\!)\ \mathrm{est}(R)$	Theorem 8.1 $\mathrm{preorder}(Q) \land S \ \mathrm{Pres}\ Q$ $\implies (\!\|\frac{F(\ni)\ S}{\ni}\ \mathrm{thin}(Q)\|\!) \sqsubseteq \frac{(\!\|S\|\!)}{\ni}\ \mathrm{thin}(Q)$
2	$\sqsubseteq \frac{(\!\|S\|\!)}{\ni}\ \mathrm{thin}(Q)\ \mathrm{est}(R)$	
3	$= \frac{(\!\|S\|\!)}{\ni}\ \mathrm{est}(R)$	

$P(\mathrm{thin}(Q))\ \mathrm{union} \sqsubseteq \mathrm{union}\ \mathrm{thin}(Q)$ (8.4)

thinning each member set, then taking the union, is a thinning of the union

$P(\mathrm{thin}(Q))\ \mathrm{union}\ \ni \sqsubseteq \ni\ \ni$

every member of the result is a member of a member set — $X \sqsubseteq \frac{S}{\ni}\ \mathrm{thin}(Q) \iff X\ \ni \sqsubseteq S \land S^\circ\ X \sqsubseteq Q^\circ\ \in$, first condition

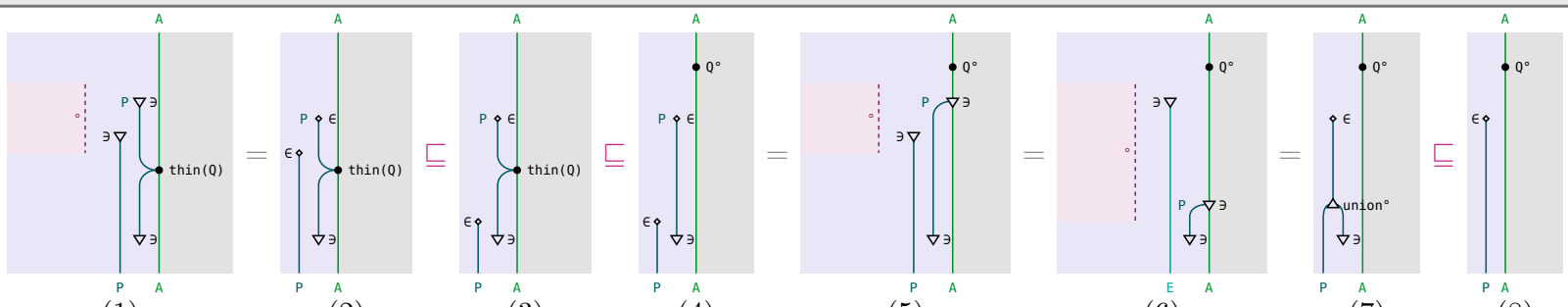


(1) (2) (3) (4)

1	$P(\mathrm{thin}(Q))\ \mathrm{union}\ \ni$	$\mathrm{union}\ \ni\ =\ \ni\ \ni$
2	$= P(\mathrm{thin}(Q))\ \ni\ \ni$	$P(R)\ \ni \sqsubseteq \ni\ R$
3	$\sqsubseteq \ni\ \mathrm{thin}(Q)\ \ni$	$\mathrm{thin}(Q)\ \ni \sqsubseteq \ni$
4	$\sqsubseteq \ni\ \ni$	

$(\ni\ \ni)^\circ\ P(\mathrm{thin}(Q))\ \mathrm{union} \sqsubseteq Q^\circ\ \in$

every member of a member set has a Q-lower bound in the result — $X \sqsubseteq \frac{S}{\ni}\ \mathrm{thin}(Q) \iff X\ \ni \sqsubseteq S \land S^\circ\ X \sqsubseteq Q^\circ\ \in$, second condition



(1) (2) (3) (4) (5) (6) (7) (8)

1	$(\ni\ \ni)^\circ\ P(\mathrm{thin}(Q))\ \mathrm{union}$	$(R\ S)^\circ = S^\circ\ R^\circ$
2	$= \in\ \in\ P(\mathrm{thin}(Q))\ \mathrm{union}$	$\in\ P(R) \sqsubseteq R\ \in$
3	$\sqsubseteq \in\ \mathrm{thin}(Q)\ \in\ \mathrm{union}$	$\in\ \mathrm{thin}(Q) \sqsubseteq Q^\circ\ \in$
4	$\sqsubseteq Q^\circ\ \in\ \in\ \mathrm{union}$	$(R\ S)^\circ = S^\circ\ R^\circ$
5	$= Q^\circ\ (\ni\ \ni)^\circ\ \mathrm{union}$	$\mathrm{union}\ \ni\ =\ \ni\ \ni$
6	$= Q^\circ\ (\mathrm{union}\ \ni)^\circ\ \mathrm{union}$	$(R\ S)^\circ = S^\circ\ R^\circ$
7	$= Q^\circ\ \in\ \mathrm{union}^\circ\ \mathrm{union}$	$\mathrm{union} = E(\ni)$
8	$\sqsubseteq Q^\circ\ \in$	

§8.2. Paths in a layered network

(8.2a)

	definition	type	meaning
1	F(B,X) = B + (B × X)	$\mathcal{A}\times\mathcal{A}\to\mathcal{A}$	a path is either one vertex a, or a vertex a followed by a path xs
2	α = [wrap,cons]	F(A,LA)→ LA	build a path: wrap one vertex as [a], or put a in front of xs
3	wrapz ≜ ⟨wrap,zero⟩	A→LA×ℕ	a path of one vertex, no edge, cost 0
4	(a,(xs,n)) ↦ ([a]+xs,wt(a,head(xs)) + n)	A×(LA×ℕ)→ LA×ℕ	put a in front of the path; the cost adds the weight of the new edge a→head(xs)
5	cost ≜ ⦇[wrapz,consw]⦈ π_2	LA→ℕ	the cost of $[a_0,\ldots,a_n]$ is $wt(a_0,a_1)+\ldots+wt(a_{n-1},a_n)$
6	R ≜ cost ≤ cost°	LA→LA	xs is cheaper than ys
7	⦇F(∋,𝟙) α⦈ = L(∋)	LA→LB	L(∋) relates a list of layers to every path that chooses one vertex from each layer
8	minpath ⊑ $\frac{L(\ni)}{\ni}$ est(R)	L(PA)→LA	the input $[x_0,\ldots,x_n]$: L(PA) is a list of layers, each a set of vertices; a path chooses one vertex from each layer, and minpath returns a cheapest one
9	Q ≜ R ∩ (head head°)	LA→LA	like R, and the two paths also start at the same vertex: head(xs) = head(ys)
10	a ↦ l(a)	B→B+C	mark a value as the left alternative
11	b ↦ r(b)	C→B+C	mark a value as the right alternative
12	cpl ≜ cp(𝟙 × A)	PB×A→ P(B×A)	pair every element of the set with the value beside it
13	cpr ≜ cp(A × 𝟙)	A×PB→ P(A×B)	pair the value beside the set with every element of the set
14	$\frac{F(\ni,\mathbb{1})}{\ni}$ = [P(l),cpl P(r)]	F(PB,X)→ PF(B,X)	cpl transpose: from layer x and a set of paths ps, choose a vertex a ∈ x
15	$\frac{F(\mathbb{1},\ni)}{\ni}$ = [$\frac{\mathbb{1}}{\ni}$ P(l),cpr P(r)]	F(B,PX)→ PF(B,X)	cpr transpose: from a vertex a and a set of paths ps, choose a tail xs ∈ ps
16	step ≜ cpr P(cons) est(R)	A×P(LA)→ LA	put a in front of every path in ps and keep a cheapest one

(8.2b)

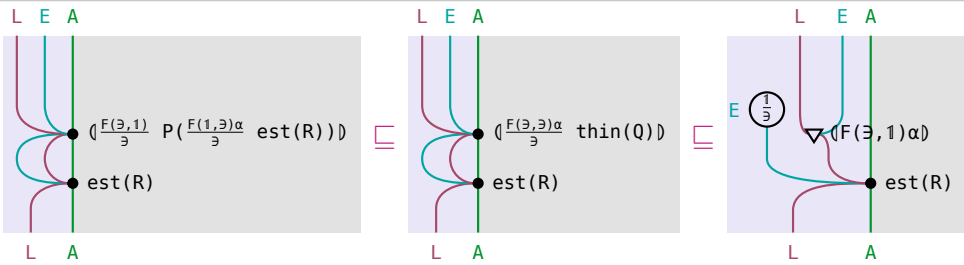
	the law	what it says
1	`F(∋,Q)α⊑F(∋,𝟙)αQ`	`F(∋,𝟙)α` is monotonic on `Q`; on `R` it is not, since the next edge can cost arbitrarily much
2	`S head⊑[𝟙,π₁]  S≜F(𝟙,∋)α`	`S head` is simple, which gives `R∩(S°S)⊑Q`: between two paths `S` builds from one argument, equal cost and equal head already means `Q`

§8.2.1. Derivation

$$(\!|\tfrac{F(\ni,1)}{\ni}\ P(\tfrac{F(1,\ni)\ \alpha}{\ni}\ \mathtt{est(R)})|\!)\ \mathtt{est(R)} \sqsubseteq \tfrac{(\!|F(\ni,1)\ \alpha|\!)}{\ni}\ \mathtt{est(R)} \tag{8.2.1a}$$

a least-cost path in a layered network, as a fold over the layers

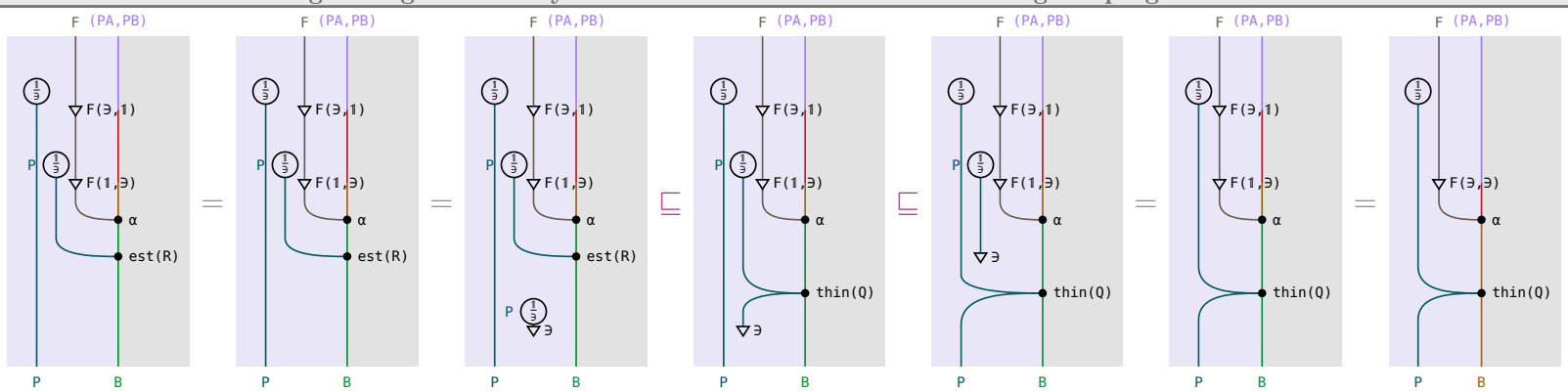
- Why it is needed: `est(R)` keeps one cheapest `w` of the set `Λ(S)(x)`; `thin(Q)` needs every dropped `z` to be `Q`-beaten by a kept one. For `{w}` alone to be a thinning we need `w Q z` for every `z` in `Λ(S)(x)`.
- We have `w R z` (`w` is cheapest) and `w (S°S) z` (both are built by `S` from the same argument `x`); the hypothesis turns the two into `w Q z`: among candidates from one argument, cheaper already means `Q`-better.



1	$(\!|\tfrac{F(\ni,1)}{\ni}\ P(\tfrac{F(1,\ni)\ \alpha}{\ni}\ \mathtt{est(R)})|\!)\ \mathtt{est(R)}$	$\tfrac{F(\ni,1)}{\ni}\ P(\tfrac{F(1,\ni)\ \alpha}{\ni}\ \mathtt{est(R)}) \sqsubseteq \tfrac{F(\ni,\ni)\ \alpha}{\ni}\ \mathtt{thin(Q)}$
2	$\sqsubseteq (\!|\tfrac{F(\ni,\ni)\ \alpha}{\ni}\ \mathtt{thin(Q)}|\!)\ \mathtt{est(R)}$	$(\!|\tfrac{F(\ni,\ni)\ \alpha}{\ni}\ \mathtt{thin(Q)}|\!)\ \mathtt{est(R)} \sqsubseteq \tfrac{(\!|F(\ni,1)\ \alpha|\!)}{\ni}\ \mathtt{est(R)}$
3	$\sqsubseteq \tfrac{(\!|F(\ni,1)\ \alpha|\!)}{\ni}\ \mathtt{est(R)}$	

$$\mathtt{S} \triangleq \mathtt{F(𝟙,\ni)\ \alpha}\ ,\ \mathtt{R} \cap (\mathtt{S}^\circ\ \mathtt{S}) \sqsubseteq \mathtt{Q} \implies \tfrac{F(\ni,1)}{\ni}\ P(\tfrac{F(1,\ni)\ \alpha}{\ni}\ \mathtt{est(R)}) \sqsubseteq \tfrac{F(\ni,\ni)\ \alpha}{\ni}\ \mathtt{thin(Q)} \tag{8.2.1b}$$

thinning the algebra of a layered network costs no more than taking the program's two cases



1	$\tfrac{F(\ni,1)}{\ni}\ P(\tfrac{F(1,\ni)\ \alpha}{\ni}\ \mathtt{est(R)})$	
2	$= \tfrac{F(\ni,1)}{\ni}\ P(\tfrac{F(1,\ni)\ \alpha}{\ni}\ \mathtt{est(R)})$	$P(\tfrac{𝟙}{\ni})\ E(\ni) = 𝟙$
3	$= \tfrac{F(\ni,1)}{\ni}\ P(\tfrac{F(1,\ni)\ \alpha}{\ni}\ \mathtt{est(R)}\ \tfrac{𝟙}{\ni})\ E(\ni)$	(8.3) $\mathtt{R} \cap (\mathtt{S}^\circ\ \mathtt{S}) \sqsubseteq \mathtt{Q}$ $\implies \tfrac{S}{\ni}\ \mathtt{est(R)}\ \tfrac{𝟙}{\ni} \sqsubseteq \tfrac{S}{\ni}\ \mathtt{thin(Q)}$
4	$\sqsubseteq \tfrac{F(\ni,1)}{\ni}\ P(\tfrac{F(1,\ni)\ \alpha}{\ni})\ P(\mathtt{thin(Q)})\ E(\ni)$	(8.4) $P(\mathtt{thin(Q)})\ \mathtt{union} \sqsubseteq \mathtt{union}\ \mathtt{thin(Q)}$
5	$\sqsubseteq \tfrac{F(\ni,1)}{\ni}\ P(\tfrac{F(1,\ni)\ \alpha}{\ni})\ E(\ni)\ \mathtt{thin(Q)}$	$\tfrac{V\ S}{\ni} = \tfrac{V}{\ni}\ P(\tfrac{S}{\ni})\ \mathtt{union}$
6	$= \tfrac{F(\ni,1)\ F(1,\ni)\ \alpha}{\ni}\ \mathtt{thin(Q)}$	$\mathtt{F(R,𝟙)\ F(𝟙,S) = F(R,S)}$
7	$= \tfrac{F(\ni,\ni)\ \alpha}{\ni}\ \mathtt{thin(Q)}$	

$= \mathtt{cp(F)\ P(\alpha)\ est(R) = [wrap,step]}$

at the layered network (`F(A,X)=A+A×X`, `B` the paths `V⁺`, `α=[wrap,cons]`): $\tfrac{F(1,\ni)}{\ni}$ `P(α) est(R)=[wrap,step]` — `step ≜ cpr P(cons) est(R)`

$= \tfrac{F(\ni,1)}{\ni}\ \mathtt{=𝟙+cpl}$ — $\tfrac{F(\ni,1)}{\ni}\ \mathtt{= [P(l),cpl\ P(r)]}$

(8.2.1c)

```
F(A,X)    =  A   +   A  ×  X
             |       |     |
             R       R     S
             ↓       ↓     ↓
F(A',X')  =  A'  +   A' ×  X'
```

`u F(R,S) v` holds iff one of:

- `u`, `v` are both `inl` and `d R d'`: the leaf moves by `R`;

- `u`, `v` are both `inr` and p_1 `R` q_1 ∧ p_2 `S` q_2: the vertex moves by `R`, the rest by `S`.
- An `inl` never relates to an `inr`: `R+S≜[R inl, S inr]` returns to the summand it came from, and `F(𝟙)=𝟙` allows no crossing.
- `F(∃,∃)` relates `inr({a,b},{p,q})` to `inr(a,p)`, `inr(a,q)`, `inr(b,p)`, `inr(b,q)`, and `inl({a,b})` to `inl(a)`, `inl(b)`.

§8.3. Implementing thin

(8.3a)

	definition	type	meaning
1	(setify ∋)(xs,x) ⟺ x ∈ xs	[A]→PA	the set of a list's elements
2	cup ≜ $\frac{(\pi_1\ \ni)\ \cup\ (\pi_2\ \ni)}{\ni}$	PA×PA→PA	the union of two sets
3	cp(F) ≜ $\frac{F(\ni)}{\ni}$	F(PA)→PFA	every F-shaped value whose parts are chosen from the given sets
4	listcpFn(zer,x) = [], listcpFn(PolyC.one,u) = [u, ()], listcpFn(const(a),c) = [c], listcpFn(arg_1,a) = [a], listcpFn(arg_2,xs) = xs, listcpFn(l ⊕ a,l(u)) = list(inl)(listcpFn(l,u)), listcpFn(a ⊕ r,r(v)) = list(inr)(listcpFn(r,v)), listcpFn(l ⊗ r,(u,v)) = cppFn(listcpFn(l,u),listcpFn(r,v))	relator(list(X))→ list(relator(X))	by the shape of F: a constant or arg_1 is a one-element list, arg_2 the list itself, a sum lists its summand, a product pairs every element of one list with every element of the other
5	linear(zer) = true, linear(one) = true, linear(const(a)) = true, linear(arg_1) = true, linear(arg_2) = true, linear(l ⊕ r) = linear(l) ∧ linear(r), linear(l ⊗ r) = linear(l) ∧ linear(r) ∧ (¬ $hasArg_2$(l) ∨ ¬$hasArg_2$(r))	P(PolyC(ι))	no product in F has the argument on both sides
6	sort(⪯) ≜ setify° ordered(⪯)	PA→LA	read a set back as one of its ⪯-ordered listings
7	(a,[]) ↦ [a] , bump(Q)((a,[b]⧺xs),ys) ⟺ Q(a,b) ∧ ys = [a]⧺xs ∨ Q(b,a) ∧ ys = [b]⧺xs ∨ ¬Q(a,b) ∧ ¬Q(b,a) ∧ ys = [a]⧺[b]⧺xs	A×[A]→[A]	put a in front of a thinned list, dropping whichever of a and the old head the other beats
8	thinlist(Q) ≜ (setify thin(Q) setify°) ∩ subseq	[A]→[A]	only drop elements, and on the set of the elements be a thinning by Q
9	T ⊑ thinlist(Q) ⟺ T ⊑ subseq ∧ T setify ⊑ setify thin(Q)		what a list implementation of thin(Q) must do: only drop elements, and thin the set
10	preorder(Q) ⟹ ⦇[nil,bump(Q)]⦈ ⊑ thinlist(Q)		an implementation: thin a list in one pass, bumping each element into the thinned rest
11	minlist(Q) ≜ setify est(Q)	[A]→A	a Q-least element of the list

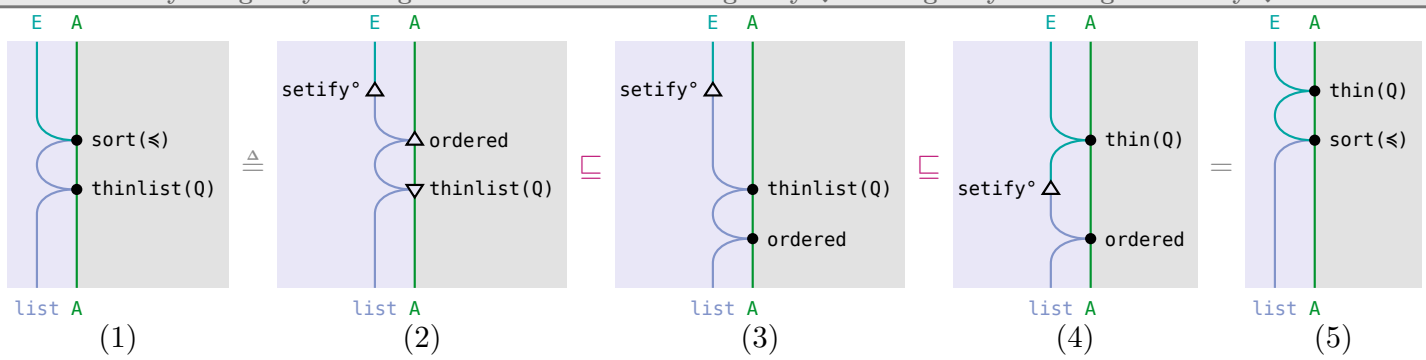
§8.3.1. Sorting sets

(8.3.1a)

	the law	what it says
1	`preorder(Q) ∧ connected(Q) ⟹ bumpFold(Q)([a]⧺xs,ys) ⟺ (∃w. minlist(Q)([a]⧺xs,w) ∧ ys = [w])` (8.5)	what thinning should come to when it can: one element
2	`sort(⩽) thinlist(Q) ⊑ thin(Q) sort(⩽)` (8.6) — (8.6) `sort(⩽) thinlist(Q) ⊑ thin(Q) sort(⩽)`	the thinning may run after the sort
3	`sort(⩽) minlist(R) ⊑ est(R)` (8.7)	a minimum of the sorted list is a minimum of the set
4	`sort(f ⩽ f°) list(f) ⊑ P(f) sort(⩽)` (8.8)	shunt a function through a sort
5	`Coreflexive(p) ⟹ sort(⩽) filter(p) ⊑ E(p) sort(⩽)` (8.9) `p : A → A`	filtering a sorted list sorts the restricted set
6	`preorder(⩽) ∧ connected(⩽) ⟹ (sort(⩽) × sort(⩽)) merge(⩽) ⊑ cup sort(⩽)` (8.10)	merging two sorted lists sorts their union
7	`F(sort(⩽)) listcp ⊑ cp(F) sort(F(⩽))` (8.11), `F(A,X)=1+A×X`	`listcp` is the list implementation of the cartesian product `cp(F)`
8	`linear(F) ⟹ relator(sort(⩽)) listcp(F,A,X) ⊑ cp(relator(F,A)) sort(relator(⩽))` (8.11), `F` polynomial and linear	the same for every linear polynomial `F`

`sort(⩽) thinlist(Q) ⊑ thin(Q) sort(⩽)` (8.6)

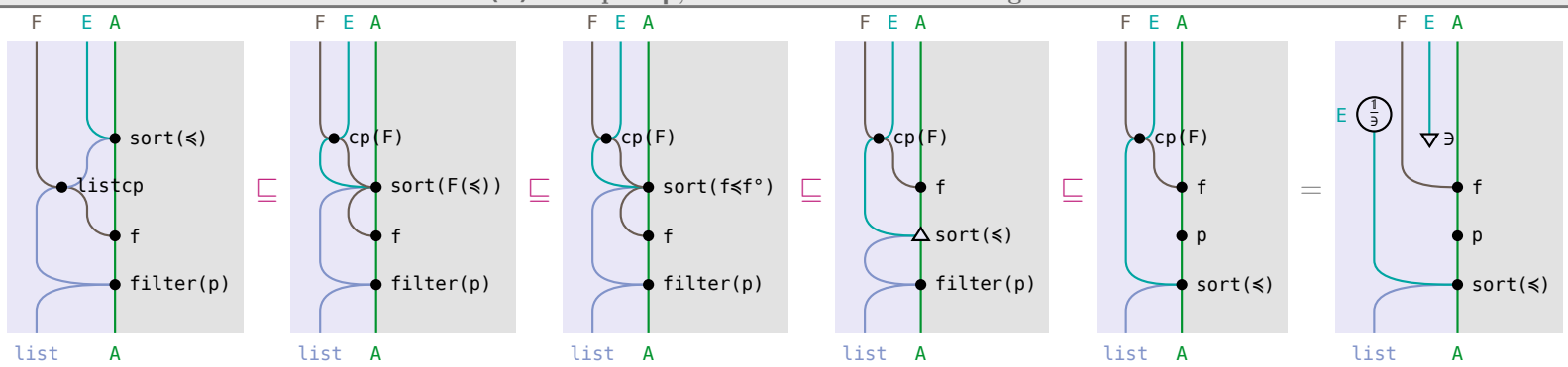
every list got by sorting the set and then thinning it by `Q` is also got by thinning the set by `Q` and then sorting.



1	`sort(⩽) thinlist(Q)`	`sort(⩽) ≜ setify° ordered(⩽)`
2	`≜ setify° ordered thinlist(Q)`	`ordered thinlist(Q) ⊑ thinlist(Q) ordered`
3	`⊑ setify° thinlist(Q) ordered`	`setify° thinlist(Q) ⊑ thin(Q) setify°`
4	`⊑ thin(Q) setify° ordered`	
5	`= thin(Q) sort(⩽)`	

Coreflexive(p) ∧ f Pres ⩽ ⟹ F(sort(⩽)) listcp list(f) filter(p) ⊑ $\frac{F(\ni)\ f\ p}{\ni}$ sort(⩽) Lemma 8.1

sorting each argument set (**F(sort(⩽))**), forming every combination of their elements (**listcp**), applying **f** to each (**list(f)**) and keeping those that pass **p** (**filter(p)**) gives one of the sorted lists of the set of all **f(u)** that pass **p**, with **u** drawn from the argument sets.



(1) (2) (3) (4) (5) (6)

1	F(sort(⩽)) listcp list(f) filter(p)	F(sort(⩽)) listcp ⊑ cp(F) sort(F(⩽))
2	⊑ cp(F) sort(F(⩽)) list(f) filter(p)	Map(f) ⟹ f Pres R ⟺ F(R) ⊑ f R f°
3	⊑ cp(F) sort(f ⩽ f°) list(f) filter(p)	sort(f ⩽ f°) list(f) ⊑ P(f) sort(⩽)
4	⊑ cp(F) P(f) sort(⩽) filter(p)	Coreflexive(p) ⟹ sort(⩽) filter(p) ⊑ E(p) sort(⩽) Map(f) ⟹ P(f) = E(f)
5	⊑ cp(F) E(f p) sort(⩽)	$\frac{S}{\ni}$ E(R) = $\frac{S\ R}{\ni}$
6	= $\frac{F(\ni)\ f\ p}{\ni}$ sort(⩽)	

(8.3.1b)

At F(A,X)=1+A×X, f(r(a,x))=a+x, ⩽ is ≤, p passes the odd numbers, input r(3,{4,1,2}).

	left side	what it does	result
1	input	an r(a,xs), xs a set	r(3,{4,1,2})
2	F(sort(⩽))	sorts the set in the X place; a is untouched	r(3,[1,2,4])
3	listcp	pairs a with each element, in list order	[r(3,1),r(3,2),r(3,4)]
4	list(f)	f on each	[4,5,7]
5	filter(p)	keeps the odd ones, order kept	[5,7]

	right side	what it does	result
1	input		r(3,{4,1,2})
2	F(∋)	relates r(3,xs) to each r(3,x) with x∈xs	r(3,4), r(3,1), r(3,2)
3	f		7, 4, 5
4	p	only odd values pass	7, 5
5	Λ(…)	collects every result into one set	{5,7}
6	sort(⩽)		[5,7]

- ⊑: the left's [5,7] is one result of the right; with ties sort(⩽) has several orders.
- f preserving ⩽ is used at list(f): [1,2,4] is sorted, so [4,5,7] is; filter(p) only drops elements. With f(r(a,x))=a−x the left gives [1,−1] after p, not sorted, so ⊑ fails.

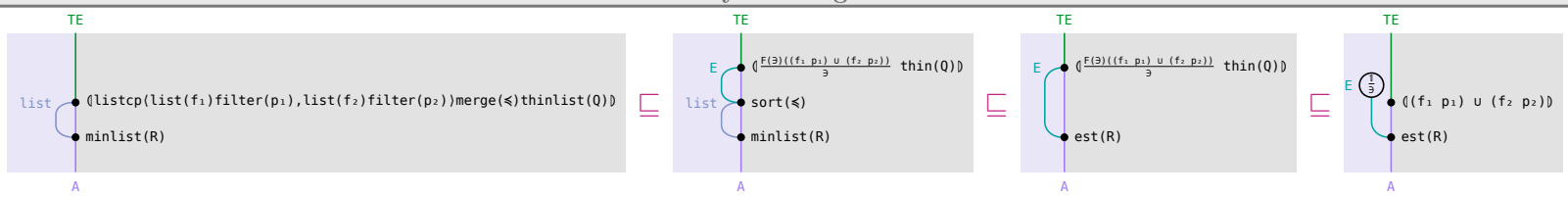
§8.3.2. Binary thinning

(8.3.2a)

Binary thinning data: Q a preorder with Q⊑R and both f_1p_1, f_2p_2 monotonic on Q; ⩽ a connected preorder with both f_1, f_2 monotonic on ⩽; R a preorder.

Coreflexive(p₁) ∧ Coreflexive(p₂) ∧ Q ⊑ R ∧ preorder(Q) ∧ preorder(R) ∧ (f₁ p₁) Pres Q ∧ (f₂ p₂) Pres Q ∧ preorder(⪯) ∧ connected(⪯) ∧ f₁ Pres ⪯ ∧ f₂ Pres ⪯ ⟹ ⦇listcp ⟨list(f₁) filter(p₁),list(f₂) filter(p₂)⟩ merge(⪯) thinlist(Q)⦈ minlist(R) ⊑ $\frac{⦇(f₁\ p₁) \cup (f₂\ p₂)⦈}{∋}$ est(R) Theorem 8.2

a fold on sorted lists of partial solutions, thinned at every step, refines the thinning specification — at binary thinning data.



(1) (2) (3) (4)

1	⦇listcp ⟨list(f₁) filter(p₁),list(f₂) filter(p₂)⟩ merge(⪯) thinlist(Q)⦈ minlist(R)	(8.3.2c) Coreflexive(p₁) ∧ Coreflexive(p₂) ∧ preorder(⪯) ∧ connected(⪯) ∧ f₁ Pres ⪯ ∧ f₂ Pres ⪯ ⟹ F(sort(⪯)) listcp ⟨list(f₁) filter(p₁),list(f₂) filter(p₂)⟩ merge(⪯) thinlist(Q) ⊑ $\frac{F(∋)\ ((f₁\ p₁) \cup (f₂\ p₂))}{∋}$ thin(Q) sort(⪯)
2	⊑ ⦇$\frac{F(∋)\ ((f₁\ p₁) \cup (f₂\ p₂))}{∋}$ thin(Q)⦈ sort(⪯) minlist(R)	sort(⪯) minlist(R) ⊑ est(R)
3	⊑ ⦇$\frac{F(∋)\ ((f₁\ p₁) \cup (f₂\ p₂))}{∋}$ thin(Q)⦈ est(R)	S₁ Pres Q ∧ S₂ Pres Q ⟹ F(Q) (S₁ ∪ S₂) ⊑ (S₁ ∪ S₂) Q
4	⊑ $\frac{⦇(f₁\ p₁) \cup (f₂\ p₂)⦈}{∋}$ est(R)	

(8.3.2b)

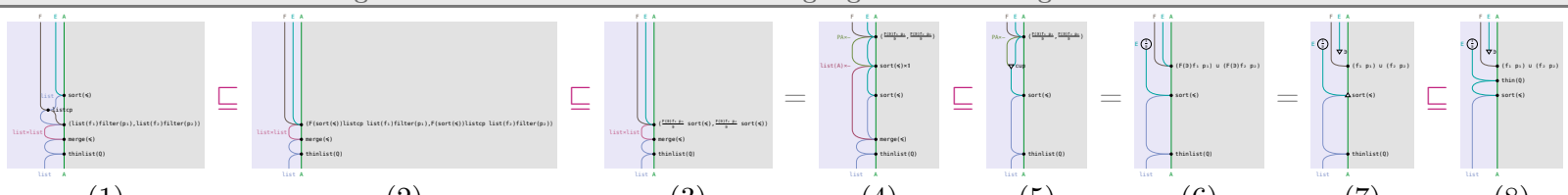
At F(A,X)=1+A×X, f₁(l())=f₂(l())=0, f₁(r(a,x))=x, f₂(r(a,x))=a+x, p₁ passes all, p₂ passes values ≤6, ⪯ is ≤, Q is =, R is ≥, input [3,3,1].

	input	list(f₁) filter(p₁)	list(f₂) filter(p₂)	merge(⪯)	thinlist(Q)
1	l()	[0]	[0]	[0,0]	[0]
2	r(1,[0])	[0]	[1]	[0,1]	[0,1]
3	r(3,[0,1])	[0,1]	[3,4]	[0,1,3,4]	[0,1,3,4]
4	r(3, [0,1,3,4])	[0,1,3,4]	[3,4,6]	[0,1,3,3,4,4,6]	[0,1,3,4,6]
5	minlist(R)				6

- merge(⪯) is one pass: both copies arrive ≤-sorted (fᵢ monotonic on ⪯, filter(pᵢ) only drops), so it only compares their two heads.
- thinlist(Q) compares neighbours only; the list is ≤-sorted, so equal values are neighbours and one pass drops every repeat.

Coreflexive(p₁) ∧ Coreflexive(p₂) ∧ preorder(⪯) ∧ connected(⪯) ∧ f₁ Pres ⪯ ∧ f₂ Pres ⪯ ⟹ F(sort(⪯)) listcp ⟨list(f₁) filter(p₁),list(f₂) filter(p₂)⟩ merge(⪯) thinlist(Q) ⊑ $\frac{F(∋)\ ((f₁\ p₁) \cup (f₂\ p₂))}{∋}$ thin(Q) sort(⪯) (8.3.2c)

sorting the candidate set turns the thinning algebra into an algebra on sorted lists.



(1) (2) (3) (4) (5) (6) (7) (8)

1	F(sort(⪯)) listcp ⟨list(f₁) filter(p₁),list(f₂) filter(p₂)⟩ merge(⪯) thinlist(Q)	X ⟨R,S⟩ ⊑ ⟨X R,X S⟩
2	⊑ (F(sort(⪯)) listcp list(f₁) filter(p₁),F(sort(⪯)) listcp list(f₂) filter(p₂)) merge(⪯) thinlist(Q)	Lemma 8.1 Coreflexive(p) ∧ f Pres ⪯ ⟹ F(sort(⪯)) listcp list(f) filter(p) ⊑ $\frac{F(∋)\ f\ p}{∋}$ sort(⪯)
3	⊑ ⟨$\frac{F(∋)\ f₁\ p₁}{∋}$ sort(⪯), $\frac{F(∋)\ f₂\ p₂}{∋}$ sort(⪯)⟩ merge(⪯) thinlist(Q)	⟨X,Y⟩ (R × S) = ⟨X R,Y S⟩
4	= ⟨$\frac{F(∋)\ f₁\ p₁}{∋}$, $\frac{F(∋)\ f₂\ p₂}{∋}$⟩ (sort(⪯) × sort(⪯)) merge(⪯) thinlist(Q)	preorder(⪯) ∧ connected(⪯) ⟹ (sort(⪯) × sort(⪯)) merge(⪯) ⊑ cup sort(⪯)
5	⊑ ⟨$\frac{F(∋)\ f₁\ p₁}{∋}$, $\frac{F(∋)\ f₂\ p₂}{∋}$⟩ cup sort(⪯) thinlist(Q)	$\frac{R \cup S}{∋}$ = ⟨$\frac{R}{∋}$, $\frac{S}{∋}$⟩ cup
6	= $\frac{(F(∋)\ f₁\ p₁) \cup (F(∋)\ f₂\ p₂)}{∋}$ sort(⪯) thinlist(Q)	R (S ∪ T) = (R S) ∪ (R T)
7	= $\frac{F(∋)\ ((f₁\ p₁) \cup (f₂\ p₂))}{∋}$ sort(⪯) thinlist(Q)	(8.6) sort(⪯) thinlist(Q) ⊑ thin(Q) sort(⪯)
8	⊑ $\frac{F(∋)\ ((f₁\ p₁) \cup (f₂\ p₂))}{∋}$ thin(Q) sort(⪯)	

§8.4. The knapsack problem

(8.4a)

	definition	type	meaning
1	total(f) ≜ list(f) sum	Int	add up f over the items of the list
2	subseq ≜ ⦇[nil,cons ∪ π₂]⦈	[A]→[A]	every subsequence of the list
3	value ≜ list(val) sum	[Item]→ Int	the total value of a packing
4	weight ≜ list(wt) sum	[Item]→ Int	the total weight of a packing
5	[nil,cons] = [nil,cons]	F[Item]→ [Item]	keep the item: the list constructor
6	within(w,xs) ⟺ weight(xs) ≤ w	[Item]→ [Item]	the packing weighs at most w
7	S ≜ ([nil,cons] within(w)) ∪ [nil,π₂]	F[Item]→ [Item]	at each item, keep it if the packing still fits, or drop it
8	R ≜ value ≥ value°	[Item]→ [Item]	x is worth at least as much as y
9	Q(val,wt) ≜ R ∩ (weight ≤ weight°)	[Item]→ [Item]	x is worth at least as much as y and weighs no more

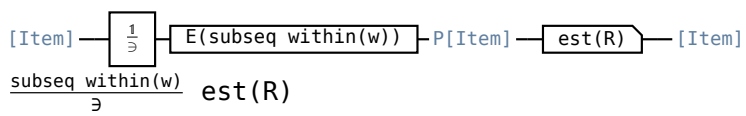
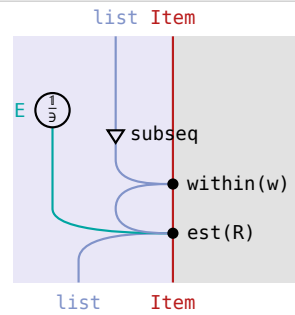
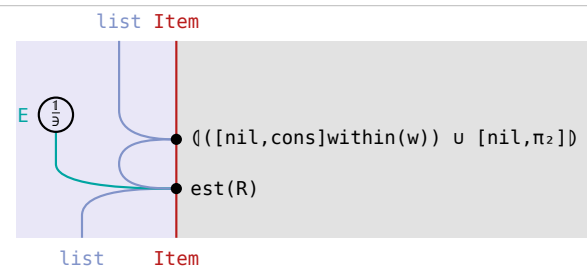
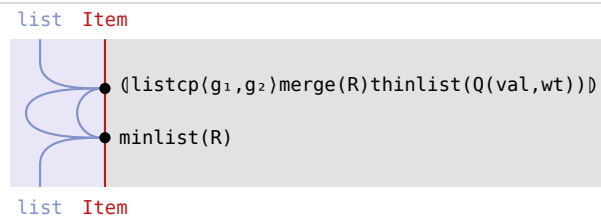
(8.4b)

	the law	what it says
1	(𝟙×R) (cons (within w))⊑cons (within w)R FALSE	not monotonic on R: a selection of greater value need not still fit once one more item goes in
2	F(Q(val,wt)) [nil,cons] within(w) ⊑ [nil,cons] within(w) Q(val,wt) F(Q(val,wt)) [nil,π₂] ⊑ [nil,π₂] Q(val,wt) ,	both halves are monotonic on Q once ties in value are broken by weight
3	F(R) [nil,cons] ⊑ [nil,cons] R F(R) [nil,π₂] ⊑ [nil,π₂] R	both algebras are monotonic on R itself, so R is the sort order P

§8.4.1. Derivation

$$0 \le w \wedge (\forall i.\ 0 \le wt(i)) \Longrightarrow (\!|\text{listcp } \langle g_1,g_2\rangle \text{ merge}(R) \text{ thinlist}(Q(val,wt))|\!) \text{ minlist}(R) \sqsubseteq \frac{\text{subseq within}(w)}{\ni} \text{ est}(R) \tag{8.4.1a}$$

the knapsack problem, as a fold that thins the packings kept at each item

circuit	Hinze–Marsden
 $\frac{\text{subseq within}(w)}{\ni}$ est(R)	
= $\frac{(\!|([nil,cons]\text{within}(w)) \cup [nil,\pi_2]|\!)}{\ni}$ est(R) fusion, weights non-negative.	
⊒ $(\!|\text{listcp } \langle g_1,g_2\rangle \text{ merge}(R) \text{ thinlist}(Q(val,wt))|\!)$ minlist(R) binary thinning theorem, at $P \triangleq R$, F linear, Q from F(Q(val,wt)) $[nil,cons]\ \text{within}(w) \sqsubseteq [nil,cons]\ \text{within}(w)\ Q(val,wt)$, $F(Q(val,wt))\ [nil,\pi_2] \sqsubseteq [nil,\pi_2]\ Q(val,wt)$	
= $(\!|[nil,\text{cpr } \langle h_1,h_2\rangle \text{ merge } R \text{ thinlist}(Q)]|\!)$ minlist(R) listcp=wrap+cpr, g_i=[list(nil),h_i] — [nil,cons] = [nil,cons] , $[nil,\pi_2](l(D)) = []$, $[nil,\pi_2](r(p)) = \pi_2(p)$ minlist(R) is head, packings coming out in descending value	

§8.5. The paragraph problem

(8.5a)

	definition	type	meaning
1	`Line ≜ L(Word)`	`Type`	a line is a non-empty list of words
2	`Para ≜ L(Line)`	`Type`	a paragraph is a non-empty list of lines
3	`new(a,xss) ≜ [[a]]⧺xss`	`Para`	start a new first line holding only the word a
4	`glue(a,[xs]) = [[a]⧺xs], glue(a,[xs]⧺a') = [[a]⧺xs]⧺a'`	`Para→Para`	put the word a at the front of the first line
5	`[wrap wrap,new](l(a)) = [[a]], [wrap wrap,new](r(q)) = new(π₁(q),π₂(q))`	`F(Para)→Para`	one word is a one-line paragraph; each further word starts a new line
6	`[wrap wrap,glue](l(a)) = [[a]], [wrap wrap,glue](r(q)) = glue(π₁(q),π₂(q))`	`F(Para)→Para`	one word is a one-line paragraph; each further word joins the first line
7	`partition ≜ ⦇[wrap wrap,new] ∪ [wrap wrap,glue]⦈`	`L(Word)→Para`	every way of breaking the words into lines
8	`width[a] = len(a), width([a]⧺xs) = (len(a) + width(xs)) + 1`	`Line→Int`	the lengths of the words plus one space between neighbours
9	`head[xs] = xs, head([xs]⧺a) = xs`	`Para→Line`	the first line of the paragraph
10	`fits(w,[xs]) = width(xs) ≤ w, fits(w,[xs]⧺a) = width(xs) ≤ w ∧ fits(w,a)`	`Para→Para`	every line is at most w wide
11	`ok(w,xss) ≜ width(head(xss)) ≤ w`	`Para→Para`	the paragraphs whose first line is at most w wide
12	`sqr(n) ≜ nn`	`Int`	n squared
13	`waste(w,[xs]) = 0, waste(w,[xs]⧺a) = sqr(w−width(xs)) + waste(w,a)`	`Para→Int`	the squared white space left on every line but the last, added up
14	`R ≜ waste(w) ≤ waste(w)°`	`Para→Para`	p wastes no more than q
15	`Q(len)(w) ≜ R ∩ (head head°)`	`Para→Para`	p wastes no more than q and has the same first line
16	`start(a) ≜ [[[a]]]`	`[Para]`	the one candidate for a single word: the paragraph of one line holding that word

(8.5b)

	the law	what it says
1	`(𝟙×R) glue⊑glue R` FALSE	`glue` is not monotonic on `R`: the waste of a paragraph depends on its whole first line, so no greedy algorithm solves this

2	`F(Q(len)(w)) [wrap wrap,new] ⊑ [wrap wrap,new] Q(len)(w)`
	`(∀a. 0 ≤ len(a)) ⟹ F(Q(len)(w)) [wrap wrap,glue] ok(w) ⊑ [wrap wrap,glue] ok(w) Q(len)(w)`
	`cons` monotonic on `collect≤collect°.` ,
		both halves are monotonic on `Q` once ties in waste are broken by the first line

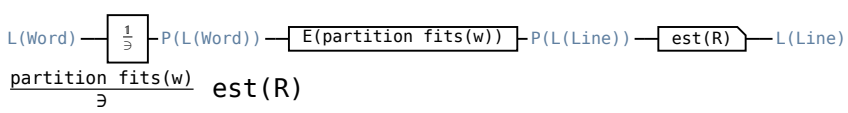
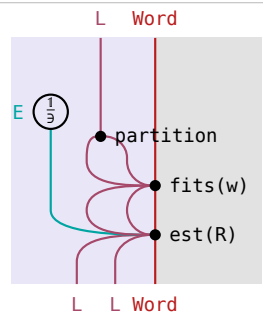
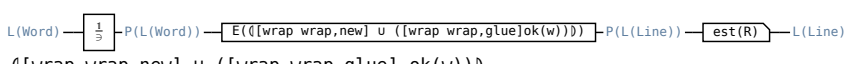
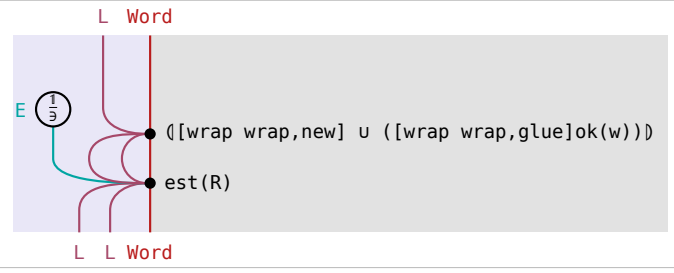
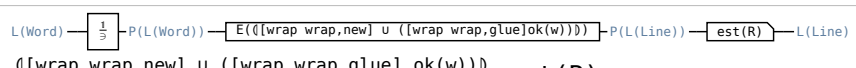
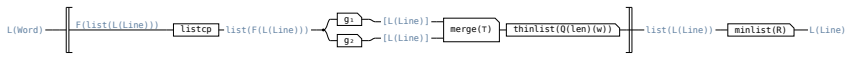
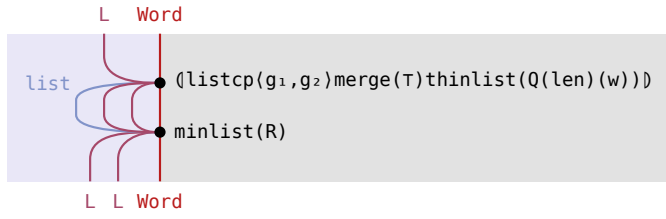
3	`merge(T) = cat`	`T` needs no sorting at all
4	`F(T) [wrap wrap,new] ⊑ [wrap wrap,new] T` `F(T) [wrap wrap,glue] ⊑ [wrap wrap,glue] T`	both algebras are monotonic on `T`, so `T` is the sort order `P`

§8.5.1. Derivation

(8.5.1a)

$$(\forall a.\ 0 \le len(a)) \wedge (\forall a.\ len(a) \le w) \Longrightarrow (\!|listcp\ \langle g_1,g_2\rangle\ merge(T)\ thinlist(Q(len)(w))|\!)$$
$$minlist(R) \sqsubseteq \frac{partition\ fits(w)}{\ni}\ est(R)$$

a paragraph laid out as a fold that thins the layouts kept at each word

circuit	Hinze–Marsden
 $\frac{partition\ fits(w)}{\ni}$ `est(R)`	
=  $\frac{⦇[wrap\ wrap,new] \cup ([wrap\ wrap,glue]\ ok(w))⦈}{\ni}$ `est(R)` fusion, every word fits on a line by itself.	
=  $\frac{⦇[wrap\ wrap,new] \cup ([wrap\ wrap,glue]\ ok(w))⦈}{\ni}$ `est(R)` the algebra as `(f₁p₁) ∪ (f₂p₂)`, `p₁≜𝟙` — binary thinning theorem.	
⊒  `⦇listcp ⟨g₁,g₂⟩ merge(T) thinlist(Q(len)(w))⦈ minlist(R)` binary thinning theorem, at `P≜T` with `merge T=cat` (`merge(T) = cat`), `Q` from `F(Q(len)(w)) [wrap wrap,new] ⊑ [wrap wrap,new] Q(len)(w)` , `(∀a. 0 ≤ len(a)) ⟹ F(Q(len)(w)) [wrap wrap,glue] ok(w) ⊑ [wrap wrap,glue] ok(w) Q(len)(w)`	
= `⦇[start,cpr ⟨h₁,h₂⟩ cat thinlist(Q)]⦈ minlist(R)` `listcp=wrap+cpr`, `gᵢ` along the coproduct — `[wrap wrap,new](l(a)) = [[a]]`, `[wrap wrap,new](r(q)) = new(π₁(q),π₂(q))` , `[wrap wrap,glue](l(a)) = [[a]]`, `[wrap wrap,glue](r(q)) = glue(π₁(q),π₂(q))`	

§8.6. Bitonic tours

(8.6a)

	definition	type	meaning
1	Journey(City) ≜ μF	Type	a list of at least two cities, the last two held as a pair
2	Tour(City) ≜ Journey(City) × Journey(City)	Type	the outward journey and the return journey
3	head(wrap(p)) = π₁(p), head([a]⧺a') = a	Journey(City)→City	the first city of the journey
4	next(wrap(p)) = π₂(p), next([a]⧺a') = head(a')	Journey(City)→City	the second city of the journey
5	[a]⧺tail(wrap(p)) = wrap(a,π₂(p)), [a]⧺tail([a']⧺a'') = [a]⧺a''	Journey(City)→Journey(City)	replace the first city of the journey by a
6	outcost(wrap(p)) = tc(p), outcost([a]⧺a') = tc(a,head(a')) + outcost(a')	Journey(City)→Int	the cost of travelling the journey forwards
7	incost(wrap(p)) = tc(π₂(p),π₁(p)), incost([a]⧺a') = tc(head(a'),a) + incost(a')	Journey(City)→Int	the cost of travelling the journey backwards
8	cost(t) ≜ outcost(π₁(t)) + incost(π₂(t))	Int	the outward journey forwards plus the return journey backwards
9	start(a) ≜ (wrap(a),wrap(a))	City×City→Tour(City)	the tour of the last two cities, both journeys being that one edge
10	dropl(a,t) ≜ ([a]⧺tail(π₁(t)),[a]⧺π₂(t))	Tour(City)	a replaces the first city of the outward journey and is put in front of the return
11	dropr(a,t) ≜ ([a]⧺π₁(t),[a]⧺tail(π₂(t)))	Tour(City)	a is put in front of the outward journey and replaces the first city of the return
12	[start,dropl](l(p)) = start(p), [start,dropl](r(q)) = dropl(π₁(q),π₂(q))	F(Tour(City))→Tour(City)	start at the last two cities, then add each city by dropl
13	[start,dropr](l(p)) = start(p), [start,dropr](r(q)) = dropr(π₁(q),π₂(q))	F(Tour(City))→Tour(City)	start at the last two cities, then add each city by dropr
14	tour ≜ ⦇[start,dropl] ∪ [start,dropr]⦈	μF→Tour(City)	every bitonic tour of the cities
15	R ≜ cost ≤ cost°	Tour(City)→Tour(City)	t costs no more than t′
16	next2(t) ≜ (next(π₁(t)),next(π₂(t)))	City×City	the second cities of both journeys
17	head2(t) ≜ (head(π₁(t)),head(π₂(t)))	City×City	the first cities of both journeys
18	Q ≜ R ∩ (next2 next2°) ∩ (head2 head2°)	Tour(City)→Tour(City)	t costs no more than t′, and both journeys of t and t′ share their first two cities

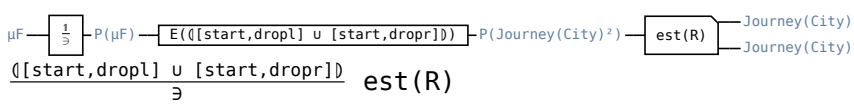
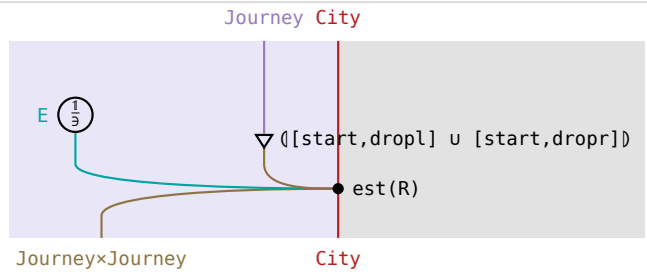
(8.6b)

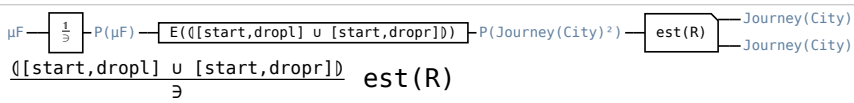
	the law	what it says
1	`(𝟙×R) dropl⊑dropl R` FALSE `(𝟙×R) dropr⊑dropr R` FALSE	neither drop is monotonic on `R`: the two edges it adds and removes depend on `head` and `next` of both lists
2	`F(Q) [start,dropl] ⊑ [start,dropl] Q` `F(Q) [start,dropr] ⊑ [start,dropr] Q` ,	both are, once ties in cost are broken by the two second cities — the heads already agree among tours of one input
3	`F(T) [start,dropl] ⊑ [start,dropl] T` `F(T) [start,dropr] ⊑ [start,dropr] T`	both algebras are monotonic on `T`, so `T` is the sort order `P`

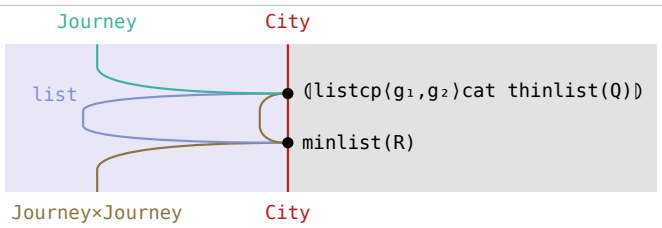
§8.6.1. Derivation

`⦇listcp ⟨g₁,g₂⟩ cat thinlist(Q)⦈ minlist(R) ⊑ ⦇[start,dropl] ∪ [start,dropr]⦈/∋ est(R)` (8.6.1a)

a least-cost bitonic tour, as a fold that thins the tours kept at each city

circuit	Hinze–Marsden
 `⦇[start,dropl] ∪ [start,dropr]⦈/∋ est(R)`	

= 
`⦇[start,dropl] ∪ [start,dropr]⦈/∋ est(R)`
`tour ≜ ⦇[start,dropl] ∪ [start,dropr]⦈`	

 `⦇listcp ⟨g₁,g₂⟩ cat thinlist(Q)⦈ minlist(R)` ⊑ binary thinning theorem, at `P≜T` with `merge T=cat`, `Q` from `F(Q) [start,dropl] ⊑ [start,dropl] Q` , `F(Q) [start,dropr] ⊑ [start,dropr] Q`	
= `⦇[start wrap,cpr ⟨list(dropl),list(dropr)⟩ cat thinlist(Q)]⦈ minlist(R)` `listcp=wrap+cpr`, `gᵢ=[list(start),list(dropᵢ)]` — `[start,dropl](l(p)) = start(p), [start,dropl](r(q)) = dropl(π₁(q),π₂(q))` , `[start,dropr](l(p)) = start(p), [start,dropr](r(q)) = dropr(π₁(q),π₂(q))` ; quadratic, two tours added per step	

§9. Dynamic Programming

§9.1. Theory

§9.1.1. Example: segmenting a list

(9.1.1a)

- `⦇T⦈ = concat`
  `T=[nil,cat]` takes a non-empty first list, so `T°` cuts a non-empty prefix off a list every way, and folding with `T` flattens a list of segments
- `⦇h⦈ = list(sum)`
  `h=[nil,cons(sum×𝟙)]` puts a segment's sum in front, so folding with `h` sums every segment
- `H = partition list(sum)`
  `H` segments a list every way, then sums each segment; with `R` comparing the largest entry, `M` cuts the list so that the largest segment sum is as small as possible

(9.1.1b)

	x	h(x)
1	l(())	[]
2	r(([c],[]))	[c]
3	r(([a, b],[c]))	[a + b, c]
4	r(([a],[b + c]))	[a, b + c]

(9.1.1c)

- `H([a, b, c],ys) ⟺ ys = [a + b + c] ∨ ys = [a, b + c] ∨ ys = [a + b, c] ∨ ys = [a, b, c]`
  `H` sends `[a,b,c]` to the sums of its four segmentations and nothing else
- `T° F(H) h = H`
  cut once, solve the rest, put the pieces back together

(9.1.1d)

	definition	type	meaning
1	`H ≜ ⦇T⦈° ⦇h⦈`	`A→B`	take the input apart by the `T`-decompositions, and put it together by `h`
2	`M ≜ ⫽H⫽ est(R)`	`A→B`	the problem to be solved: an answer of `H` that is best under `R`

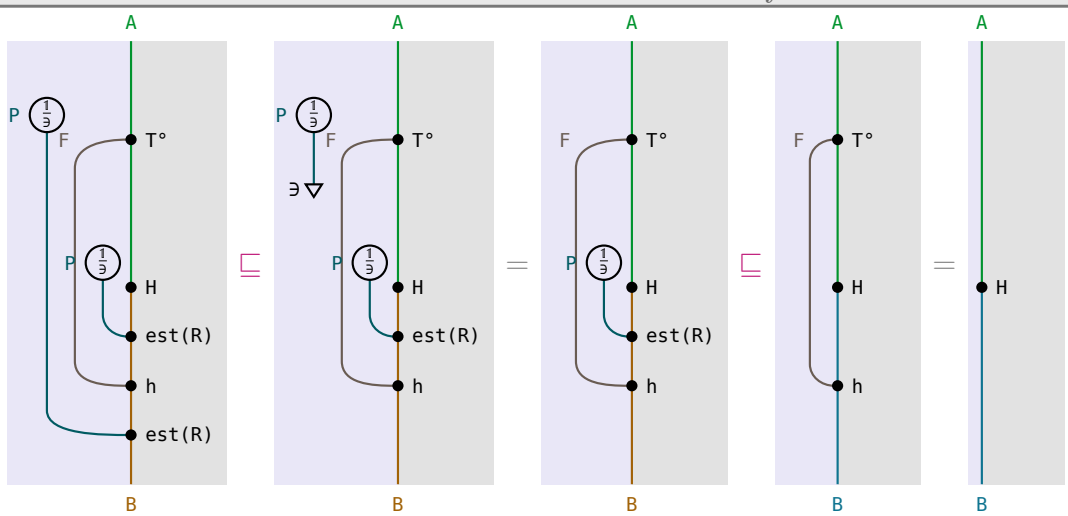
`Map(h) ∧ h Pres R° ∧ R° R° ⊑ R° ⟹ (μX : ⫽T°⫽ P(F(X) h) est(R)) ⊑ ⫽H⫽ est(R)` Theorem 9.1

every answer the recursion returns is an optimal one

- `⫽T°⫽` takes the input apart one step every way; `F(X)` solves each part by the recursion `X`; `h` assembles each candidate; `est(R)` keeps a best one
- `⊑M`: that never misses the optimum, though all candidates are never listed as `H` would
- `h` monotonic on `R`: a better part never makes the whole worse
- `⊑`, not `=`: some optima may not be returned; that the recursion returns anything at all needs `T°` to stop taking apart — dynamic programming theorem

$$\frac{T^\circ}{\ni}\ P(F(\frac{H}{\ni}\ est(R))\ h)\ est(R) \sqsubseteq H \tag{9.2}$$

taking the input apart every way **T** allows ($\frac{T^\circ}{\ni}$), solving each part by **M** and keeping an optimum (**P(F(M)h) est(R)**) returns only what **H** returns

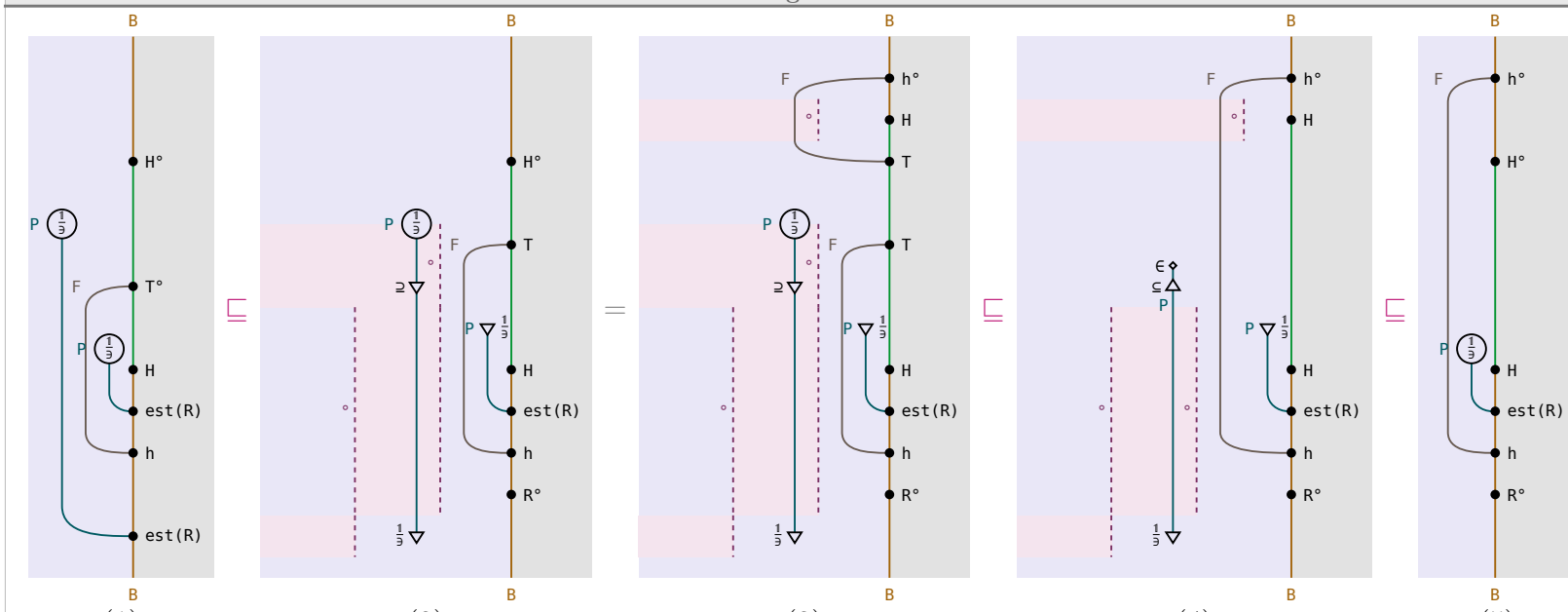


(1) (2) (3) (4) (5)

1	$\frac{T^\circ}{\ni}$ P(F($\frac{H}{\ni}$ est(R)) h) est(R)	P(S) est(R) ⊑ ∋ S
2	⊑ $\frac{T^\circ}{\ni}$ ∋ F($\frac{H}{\ni}$ est(R)) h	$\frac{R}{\ni}$ ∋ = R
3	= T° F($\frac{H}{\ni}$ est(R)) h	$\frac{S}{\ni}$ est(R) ⊑ S
4	⊑ T° F(H) h	T° F(H) h = H
5	= H	

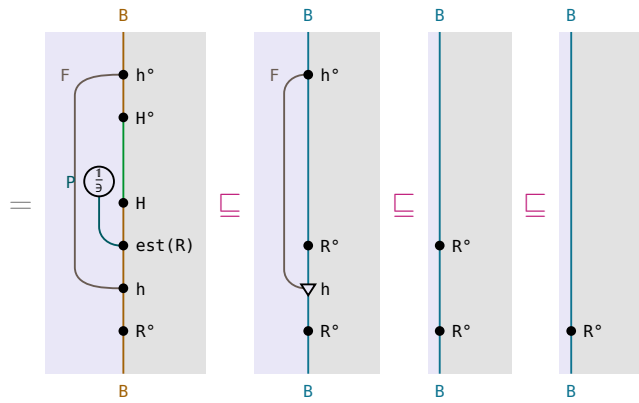
$$\texttt{Map(h)} \wedge \texttt{h Pres R°} \wedge \texttt{R° R°} \sqsubseteq \texttt{R°} \implies \texttt{H°} \ \tfrac{\texttt{T°}}{\ni}\ \texttt{P(F(}\tfrac{\texttt{H}}{\ni}\ \texttt{est(R)) h) est(R)} \sqsubseteq \texttt{R°} \tag{9.3}$$

for every **b** that **H** returns from an input, the step $\frac{\texttt{T°}}{\ni}$ **P(F(M)h) est(R)** returns from that input only **b'** with **R** relating **b'** to **b**



(1) (2) (3) (4) (5)

1	$\texttt{H°}\ \frac{\texttt{T°}}{\ni}\ \texttt{P(F(}\frac{\texttt{H}}{\ni}\ \texttt{est(R)) h) est(R)}$	$\texttt{P(S) est(R)} \sqsubseteq \in \setminus \texttt{(S R°)}$
2	$\sqsubseteq \texttt{H°}\ \frac{\texttt{T°}}{\ni}\ \texttt{(}\in \setminus \texttt{(F(}\frac{\texttt{H}}{\ni}\ \texttt{est(R)) h R°))}$	$\texttt{H° = h° F(H°) T}$
3	$= \texttt{h° F(H°) T}\ \frac{\texttt{T°}}{\ni}\ \texttt{(}\in \setminus \texttt{(F(}\frac{\texttt{H}}{\ni}\ \texttt{est(R)) h R°))}$	$\texttt{T}\ \frac{\texttt{T°}}{\ni} \sqsubseteq \in$
4	$\sqsubseteq \texttt{h° F(H°)} \in \texttt{(}\in \setminus \texttt{(F(}\frac{\texttt{H}}{\ni}\ \texttt{est(R)) h R°))}$	$\texttt{S (S} \setminus \texttt{R)} \sqsubseteq \texttt{R}$
5	$\sqsubseteq \texttt{h° F(H°) F(}\frac{\texttt{H}}{\ni}\ \texttt{est(R)) h R°}$	



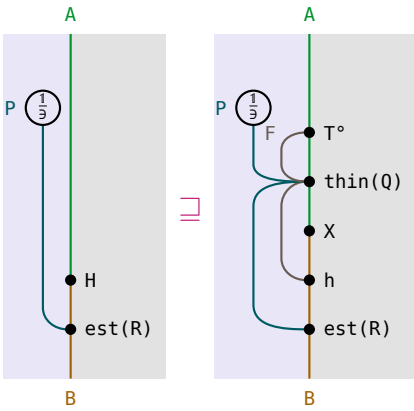
(6) (7) (8) (9)

6	$= \texttt{h° F(H°}\ \frac{\texttt{H}}{\ni}\ \texttt{est(R)) h R°}$	$\texttt{S°}\ \frac{\texttt{S}}{\ni}\ \texttt{est(R)} \sqsubseteq \texttt{R°}$
7	$\sqsubseteq \texttt{h° F(R°) h R°}$	$\texttt{Map(f)}$ $\implies \texttt{f Pres R} \iff \texttt{f° F(R) f} \sqsubseteq \texttt{R}$
8	$\sqsubseteq \texttt{R° R°}$	$\texttt{R° R°} \sqsubseteq \texttt{R°}$
9	$\sqsubseteq \texttt{R°}$	

$\mathsf{Map(h)} \wedge \mathsf{h\ Pres\ R^\circ} \wedge \mathsf{R^\circ\ R^\circ} \sqsubseteq \mathsf{R^\circ} \wedge \mathsf{ThinCondition(T,h,R,Q)} \Longrightarrow (\mu X : \frac{T^\circ}{\ni}\ \mathsf{thin(Q)\ P(F(X)\ h)\ est(R)}) \sqsubseteq \frac{H}{\ni}\ \mathsf{est(R)}$ Theorem 9.2

an optimum over everything H returns is reached by taking the input apart every way T allows, dropping the parts that can never win, solving each of the rest and keeping one optimum

$H \triangleq (\!|T|\!)^\circ (\!|h|\!)$ — $H \triangleq (\!|T|\!)^\circ\ (\!|h|\!)$



(1) (2)

1	$\frac{H}{\ni}$ est(R)	$QF(H)h \sqsubseteq F(H)hR$, h monotonic on R — $Q \sqsubseteq R \wedge \mathbb{1} \sqsubseteq Q \wedge R\ R \sqsubseteq R$ $\Longrightarrow$ est(R) = thin(Q) est(R) , $X \sqsubseteq$ est(R) $\Longleftrightarrow X \sqsubseteq \ni \wedge \in X \sqsubseteq R^\circ$
2	$\sqsupseteq \frac{T^\circ}{\ni}$ thin(Q) P(F(X) h) est(R)	

$$\begin{tikzcd}
FA \arrow[r, "Q"] \arrow[d, "F(H)"'] & FA \arrow[d, "F(H)"] \\
FB \arrow[d, "h"'] \arrow[r, phantom, "\sqsubseteq"] & FB \arrow[d, "h"] \\
B \arrow[r, "R"'] & B
\end{tikzcd}$$

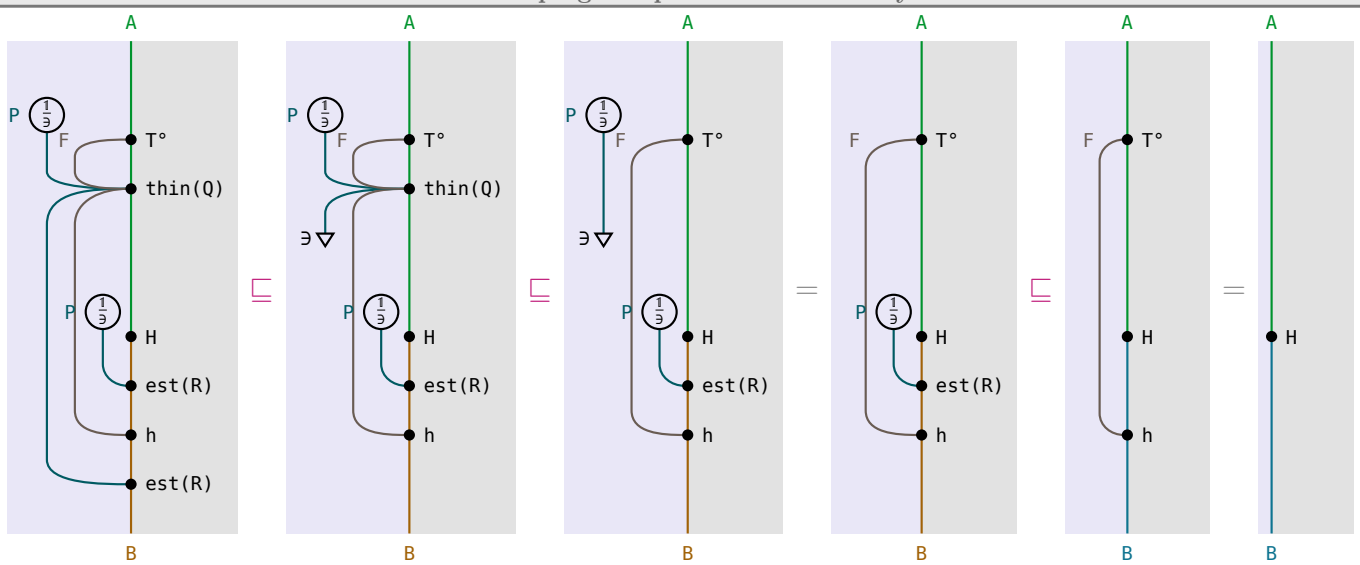
$$\begin{tikzcd}
A \arrow[r, "T^\circ"] \arrow[d, "H"'] & FA \arrow[d, "F(H)"] \\
B & FB \arrow[l, "h"]
\end{tikzcd}$$

$T^\circ F(H)h=H$: taking the input apart once by T°, solving the parts by F(H) and assembling by h returns exactly what H returns

$QF(H)h \sqsubseteq F(H)hR$: every result of thinning by Q, solving the parts by F(H) and assembling by h is reached by R from a result of solving by F(H) and assembling by h without Q

$\frac{T^\circ}{\ni}$ thin(Q) P(F($\frac{H}{\ni}$ est(R)) h) est(R) $\sqsubseteq$ H (9.1.1e)

taking the input apart every way T allows ($\frac{T^\circ}{\ni}$), dropping the parts thin(Q) rejects, solving each rest by M and keeping an optimum returns only what H returns

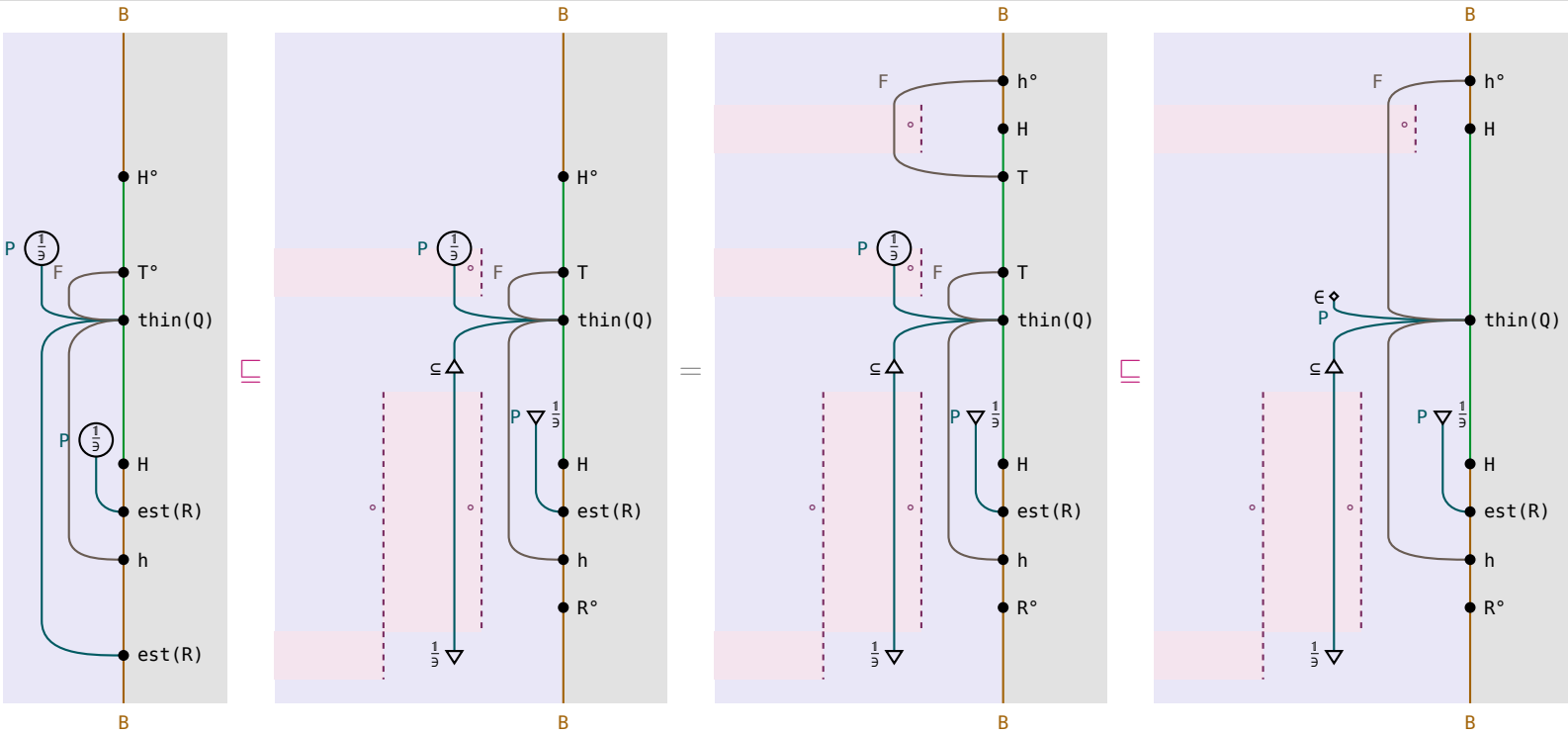


(1) (2) (3) (4) (5) (6)

1	$\frac{T^\circ}{\ni}$ thin(Q) P(F($\frac{H}{\ni}$ est(R)) h) est(R)	P(S) est(R) $\sqsubseteq \ni$ S
2	$\sqsubseteq \frac{T^\circ}{\ni}$ thin(Q) $\ni$ F($\frac{H}{\ni}$ est(R)) h	thin(Q) $\ni \sqsubseteq \ni$
3	$\sqsubseteq \frac{T^\circ}{\ni} \ni$ F($\frac{H}{\ni}$ est(R)) h	$\frac{R}{\ni} \ni = R$
4	$= T^\circ$ F($\frac{H}{\ni}$ est(R)) h	$\frac{S}{\ni}$ est(R) $\sqsubseteq$ S
5	$\sqsubseteq T^\circ$ F(H) h	T° F(H) h = H
6	$=$ H	

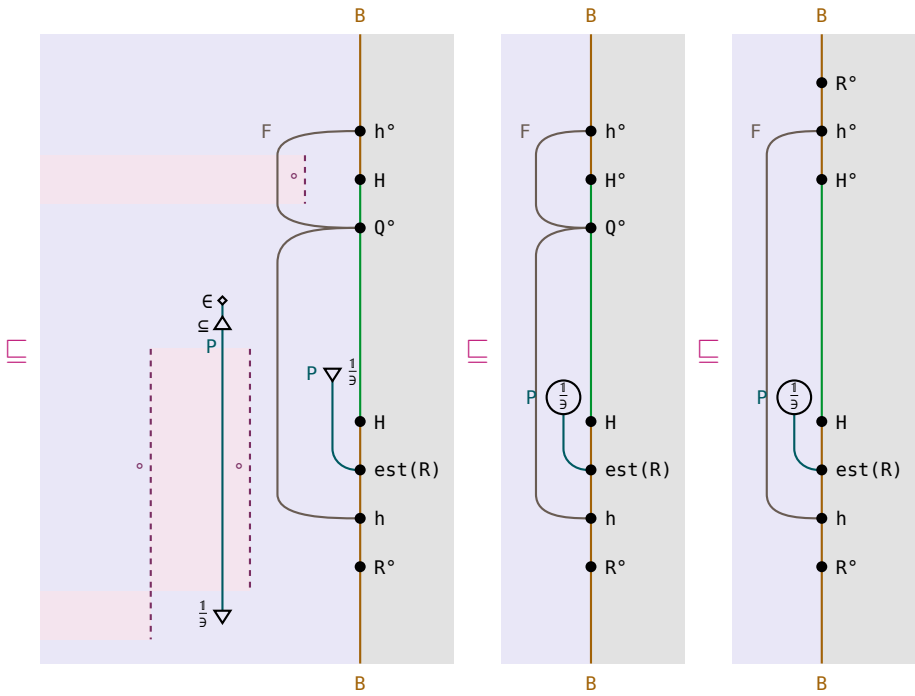
Map(h) ∧ h Pres R° ∧ R° R° ⊑ R° ∧ ThinCondition(T,h,R,Q) ⟹ H° $\frac{T°}{ɜ}$ thin(Q) P(F($\frac{H}{ɜ}$ est(R)) h) est(R) ⊑ R° (9.1.1f)

H° followed by the body at M is ⊑R°: an answer of the body is never worse than an answer of H to the same input



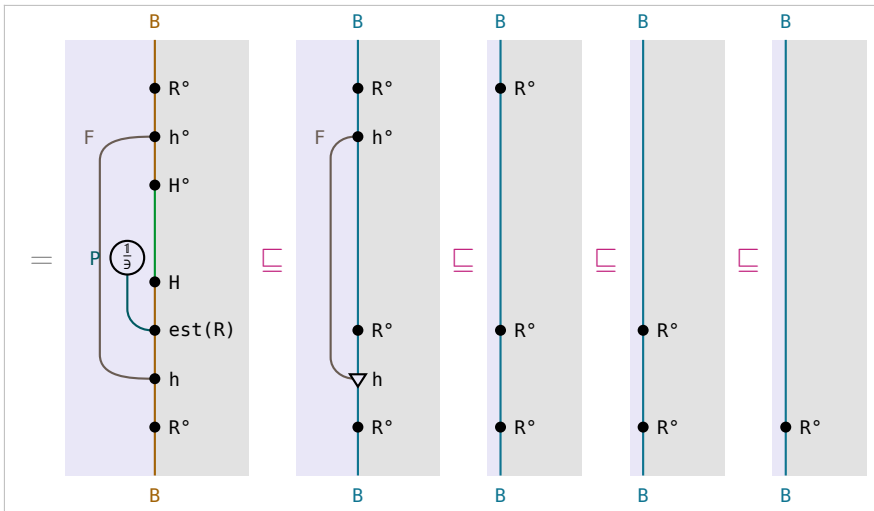
(1) (2) (3) (4)

1	H° $\frac{T°}{ɜ}$ thin(Q) P(F($\frac{H}{ɜ}$ est(R)) h) est(R)	P(S) est(R) ⊑ ∈ \ (S R°)
2	⊑ H° $\frac{T°}{ɜ}$ thin(Q) (∈ \ (F($\frac{H}{ɜ}$ est(R)) h R°))	H° = h° F(H°) T
3	= h° F(H°) T $\frac{T°}{ɜ}$ thin(Q) (∈ \ (F($\frac{H}{ɜ}$ est(R)) h R°))	T $\frac{T°}{ɜ}$ ⊑ ∈
4	⊑ h° F(H°) ∈ thin(Q) (∈ \ (F($\frac{H}{ɜ}$ est(R)) h R°))	∈ thin(Q) ⊑ Q° ∈



(5) (6) (7)

5	⊑ h° F(H°) Q° ∈ (∈ \ (F($\frac{H}{ɜ}$ est(R)) h R°))	S (S \ R) ⊑ R
6	⊑ h° F(H°) Q° F($\frac{H}{ɜ}$ est(R)) h R°	ThinCondition(T,h,R,Q) ⟹ h° F(H°) Q° ⊑ R° h° F(H°)
7	⊑ R° h° F(H°) F($\frac{H}{ɜ}$ est(R)) h R°	



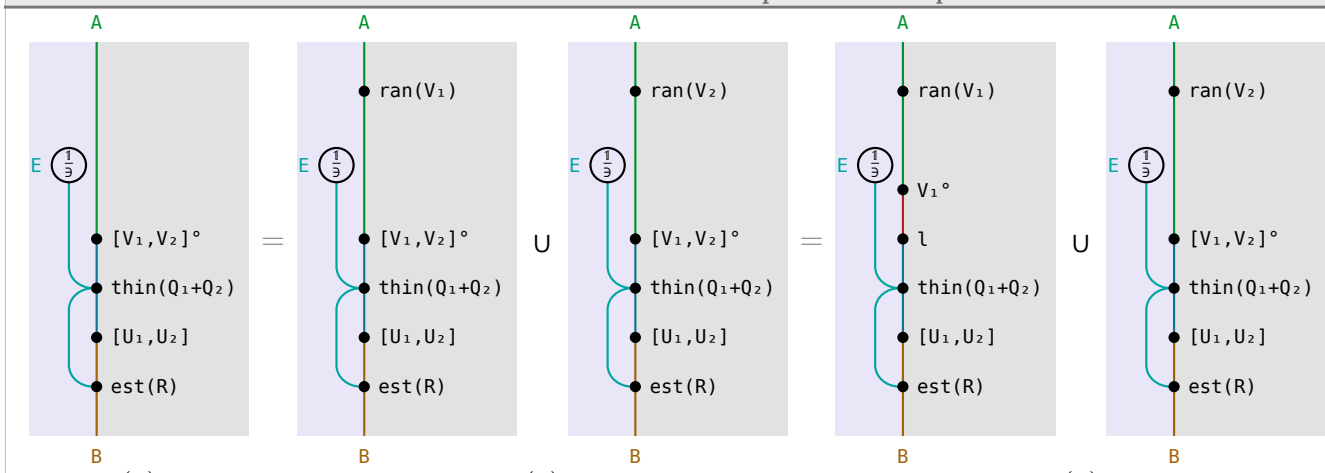
8	= R° h° F(H° $\frac{H}{\ni}$ est(R)) h R°	S° $\frac{S}{\ni}$ est(R) ⊑ R°
9	⊑ R° h° F(R°) h R°	Map(f) ⟹ f Pres R ⟺ f° F(R) f ⊑ R
10	⊑ R° R° R°	R° R° ⊑ R°
11	⊑ R° R°	R° R° ⊑ R°
12	⊑ R°	

§9.1.2. Checking the conditions

Proposition 9.1

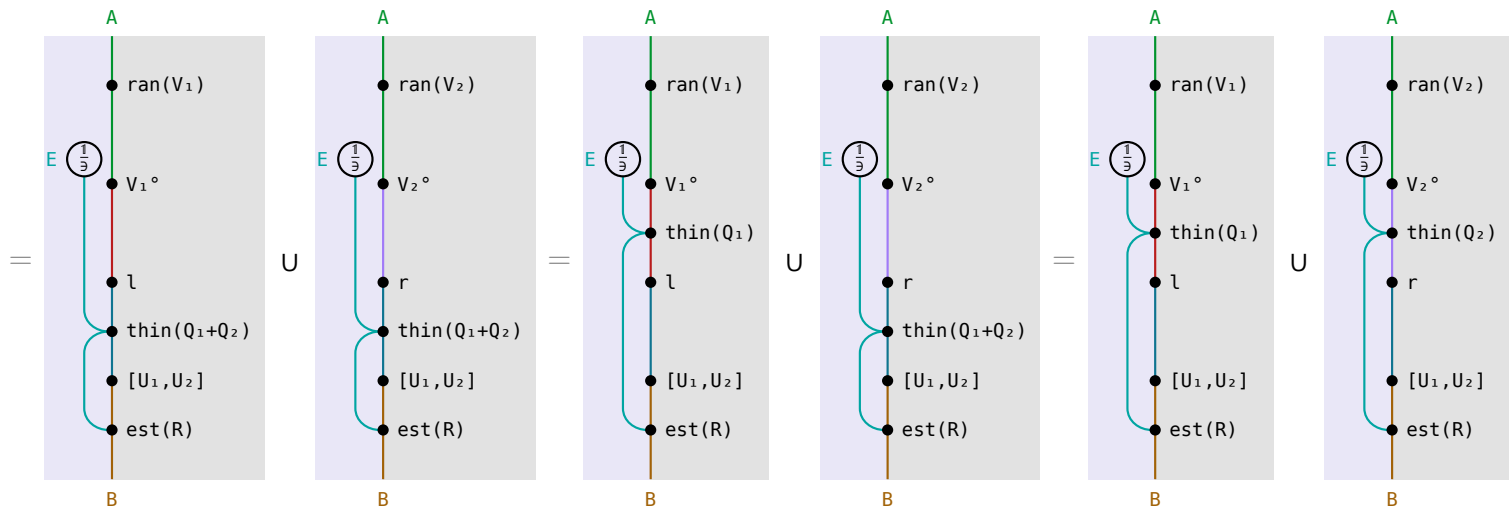
$$V_2\ V_1{}^\circ = \emptyset \implies \frac{[V_1,V_2]^\circ}{\ni}\ \mathrm{thin}(Q_1 + Q_2)\ P([U_1,U_2])\ \mathrm{est}(R) = (\mathrm{ran}(V_1)\ \frac{V_1{}^\circ}{\ni}\ \mathrm{thin}(Q_1)\ P(U_1)\ \mathrm{est}(R)) \cup (\mathrm{ran}(V_2)\ \frac{V_2{}^\circ}{\ni}\ \mathrm{thin}(Q_2)\ P(U_2)\ \mathrm{est}(R))$$

when V_1 and V_2 have disjoint ranges, the body over $[V_1,V_2]$ runs the V_1 problem on inputs V_1 reaches and the V_2 problem on inputs V_2 reaches



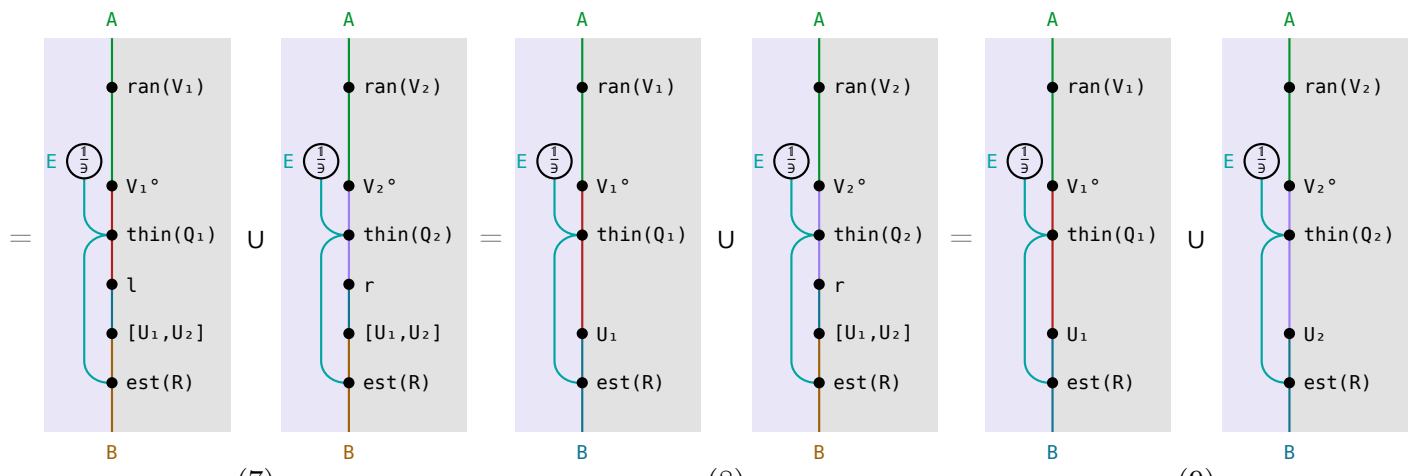
(1) (2) (3)

1	$\frac{[V_1,V_2]^\circ}{\ni}\ \mathrm{thin}(Q_1 + Q_2)\ P([U_1,U_2])\ \mathrm{est}(R)$	$\frac{[V_1,V_2]^\circ}{\ni}\ \mathrm{thin}(Q_1 + Q_2)\ P([U_1,U_2])\ \mathrm{est}(R) = (\mathrm{ran}(V_1)\ \frac{[V_1,V_2]^\circ}{\ni}\ \mathrm{thin}(Q_1 + Q_2)\ P([U_1,U_2])\ \mathrm{est}(R)) \cup (\mathrm{ran}(V_2)\ \frac{[V_1,V_2]^\circ}{\ni}\ \mathrm{thin}(Q_1 + Q_2)\ P([U_1,U_2])\ \mathrm{est}(R))$
2	$= (\mathrm{ran}(V_1)\ \frac{[V_1,V_2]^\circ}{\ni}\ \mathrm{thin}(Q_1 + Q_2)\ P([U_1,U_2])\ \mathrm{est}(R)) \cup (\mathrm{ran}(V_2)\ \frac{[V_1,V_2]^\circ}{\ni}\ \mathrm{thin}(Q_1 + Q_2)\ P([U_1,U_2])\ \mathrm{est}(R))$	$V_2\ V_1{}^\circ = \emptyset \implies \mathrm{ran}(V_1)\ \frac{[V_1,V_2]^\circ}{\ni} = \mathrm{ran}(V_1)\ \frac{V_1{}^\circ}{\ni}\ P(l)$
3	$= (\mathrm{ran}(V_1)\ \frac{V_1{}^\circ}{\ni}\ P(l)\ \mathrm{thin}(Q_1 + Q_2)\ P([U_1,U_2])\ \mathrm{est}(R)) \cup (\mathrm{ran}(V_2)\ \frac{[V_1,V_2]^\circ}{\ni}\ \mathrm{thin}(Q_1 + Q_2)\ P([U_1,U_2])\ \mathrm{est}(R))$	$V_2\ V_1{}^\circ = \emptyset \implies \mathrm{ran}(V_2)\ \frac{[V_1,V_2]^\circ}{\ni} = \mathrm{ran}(V_2)\ \frac{V_2{}^\circ}{\ni}\ P(r)$



(4) (5) (6)

4	$= (\mathrm{ran}(V_1)\ \frac{V_1{}^\circ}{\ni}\ P(l)\ \mathrm{thin}(Q_1 + Q_2)\ P([U_1,U_2])\ \mathrm{est}(R)) \cup (\mathrm{ran}(V_2)\ \frac{V_2{}^\circ}{\ni}\ P(r)\ \mathrm{thin}(Q_1 + Q_2)\ P([U_1,U_2])\ \mathrm{est}(R))$	$P(l)\ \mathrm{thin}(Q_1 + Q_2) = \mathrm{thin}(Q_1)\ P(l)$
5	$= (\mathrm{ran}(V_1)\ \frac{V_1{}^\circ}{\ni}\ \mathrm{thin}(Q_1)\ P(l)\ P([U_1,U_2])\ \mathrm{est}(R)) \cup (\mathrm{ran}(V_2)\ \frac{V_2{}^\circ}{\ni}\ P(r)\ \mathrm{thin}(Q_1 + Q_2)\ P([U_1,U_2])\ \mathrm{est}(R))$	$P(r)\ \mathrm{thin}(Q_1 + Q_2) = \mathrm{thin}(Q_2)\ P(r)$
6	$= (\mathrm{ran}(V_1)\ \frac{V_1{}^\circ}{\ni}\ \mathrm{thin}(Q_1)\ P(l)\ P([U_1,U_2])\ \mathrm{est}(R)) \cup (\mathrm{ran}(V_2)\ \frac{V_2{}^\circ}{\ni}\ \mathrm{thin}(Q_2)\ P(r)\ P([U_1,U_2])\ \mathrm{est}(R))$	



(7) (8) (9)

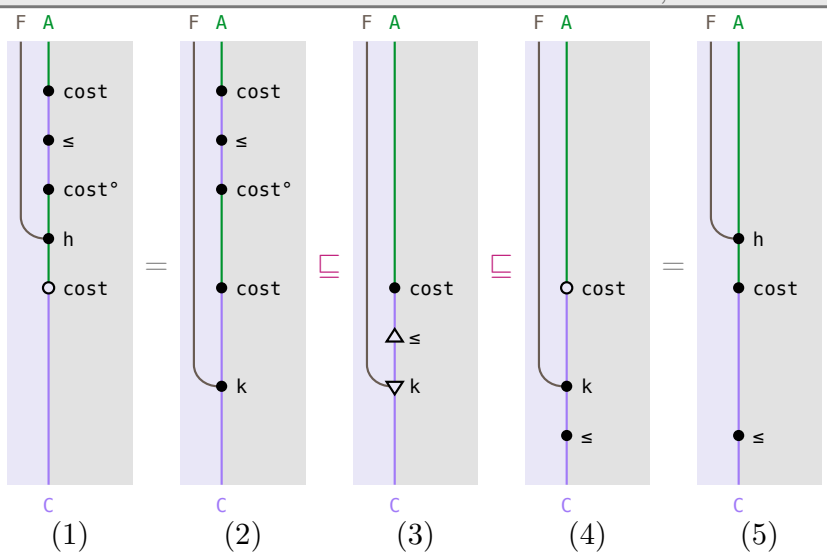
7	$= (\mathrm{ran}(V_1)\ \frac{V_1{}^\circ}{\ni}\ \mathrm{thin}(Q_1)\ P(l\ [U_1,U_2])\ \mathrm{est}(R)) \cup (\mathrm{ran}(V_2)\ \frac{V_2{}^\circ}{\ni}\ \mathrm{thin}(Q_2)\ P(r\ [U_1,U_2])\ \mathrm{est}(R))$	$l\ [R,S] = R$
8	$= (\mathrm{ran}(V_1)\ \frac{V_1{}^\circ}{\ni}\ \mathrm{thin}(Q_1)\ P(U_1)\ \mathrm{est}(R)) \cup (\mathrm{ran}(V_2)\ \frac{V_2{}^\circ}{\ni}\ \mathrm{thin}(Q_2)\ P(r\ [U_1,U_2])\ \mathrm{est}(R))$	$r\ [R,S] = S$
9	$= (\mathrm{ran}(V_1)\ \frac{V_1{}^\circ}{\ni}\ \mathrm{thin}(Q_1)\ P(U_1)\ \mathrm{est}(R)) \cup (\mathrm{ran}(V_2)\ \frac{V_2{}^\circ}{\ni}\ \mathrm{thin}(Q_2)\ P(U_2)\ \mathrm{est}(R))$	

Map(cost) ∧ h cost = F(cost) k ∧ F(≤) k ⊑ k ≤ ⟹ F(cost ≤ cost°) h ⊑ h cost ≤ cost° Proposition 9.2

if **h** then **cost** equals **F(cost)** then a **k** monotonic on **≤**, then **h** is monotonic on **cost≤cost°**, which compares two values by comparing their **cost**s under **≤**

Map(cost) ∧ h cost = F(cost) k ∧ F(≤) k ⊑ k ≤ ⟹ F(cost ≤ cost°) h cost ⊑ h cost ≤

by shunting at the map **cost**, **F(cost≤cost°)h⊑h cost≤cost°** is **F(cost≤cost°)h cost⊑h cost ≤**, which the chain proves



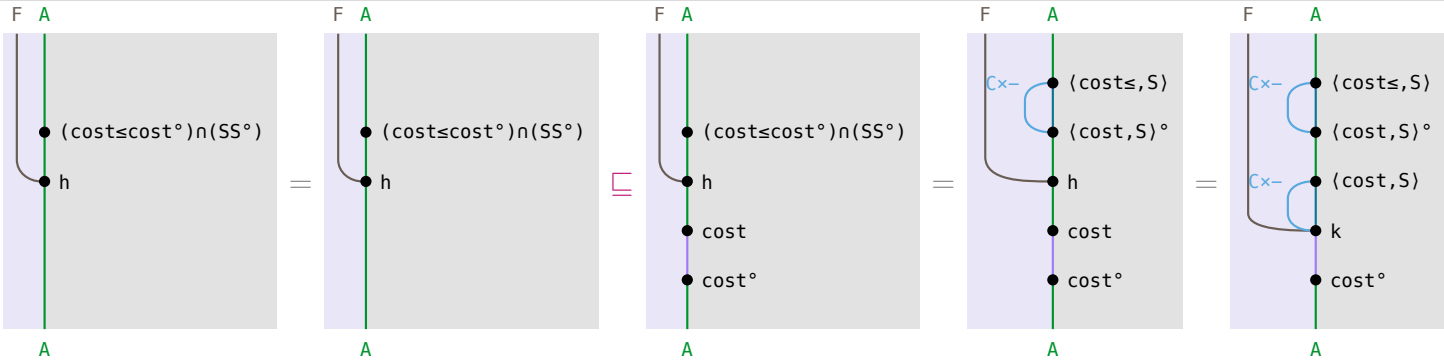
1	F(cost ≤ cost°) h cost	h cost = F(cost) k
2	= F(cost ≤ cost°) F(cost) k	Map(f) ∧ R = f S f° ⟹ R f ⊑ f S
3	⊑ F(cost) F(≤) k	F(≤) k ⊑ k ≤
4	⊑ F(cost) k ≤	h cost = F(cost) k
5	= h cost ≤	

(9.1.2a)

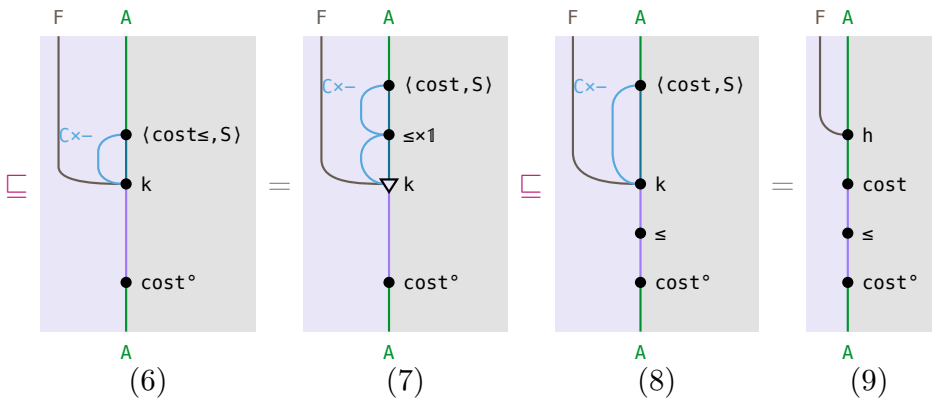
	the condition	how it is discharged
1	F(R)h⊑hR Proposition 9.2, R≜cost≤cost°, h cost=F(cost)k, F(≤)k⊑k≤	monotonicity when the cost is itself a fold with a step k monotonic on ≤
2	F(R∩(H°H))h⊑hR Proposition 9.3, R≜cost≤cost°, h cost=F(⟨cost,H°⟩)k, F(≤×𝟙)k⊑k≤, H° simple;	monotonicity **in context**: k may also read the input the part was built from
3	QF(H)h⊑F(H)hR at Q≜F(U,V) Proposition 9.4, U, V preorders, F(U,R)h⊑hR, VH⊑HR	both conditions at once, split along the two arguments of a bifunctor

Map(cost) ∧ Simple(S) ∧ h cost = F(⟨cost,S⟩) k ∧ F(≤ × 𝟙) k ⊑ k ≤ ⟹ F((cost ≤ cost°) ∩ (S S°)) h ⊑ h cost ≤ cost° Proposition 9.3

improving each part by **R** within its **S**-context, then assembling by **h**, is below **hR**



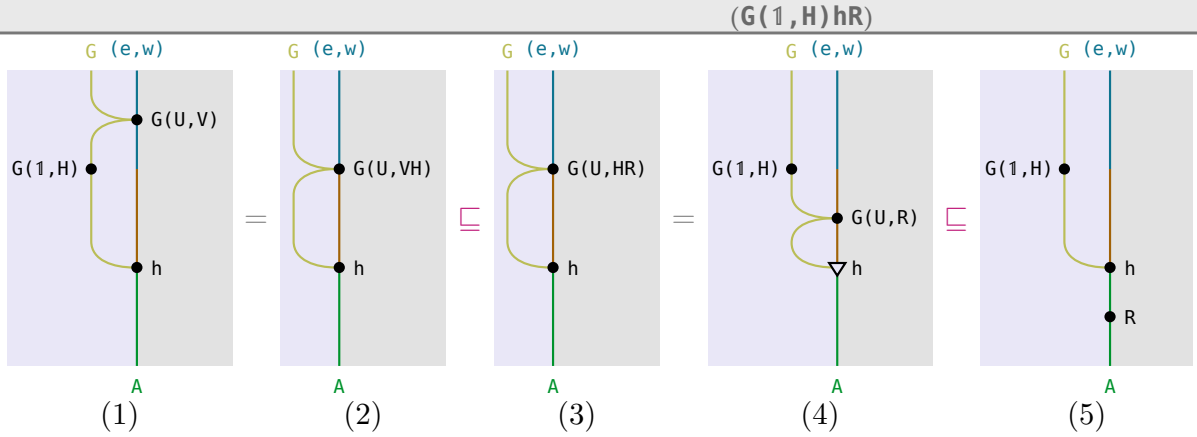
1	F((cost ≤ cost°) ∩ (S S°)) h	
2	= F((cost ≤ cost°) ∩ (S S°)) h	Map(f) ⟹ 𝟙 ⊑ f f°
3	⊑ F((cost ≤ cost°) ∩ (S S°)) h cost cost°	⟨X,Y⟩ ⟨R,S⟩° = (X R°) ∩ (Y S°)
4	= F(⟨cost ≤,S⟩ ⟨cost,S⟩°) h cost cost°	h cost = F(⟨cost,S⟩) k
5	= F(⟨cost ≤,S⟩ ⟨cost,S⟩° ⟨cost,S⟩) k cost°	



6	⊑ F(⟨cost ≤,S⟩) k cost°	⟨X,Y⟩ (R × 𝟙) = ⟨X R,Y⟩
7	= F(⟨cost,S⟩) F(≤ × 𝟙) k cost°	F(≤ × 𝟙) k ⊑ k ≤
8	⊑ F(⟨cost,S⟩) k ≤ cost°	h cost = F(⟨cost,S⟩) k
9	= h cost ≤ cost°	

G(U,R) h ⊑ h R ∧ V H ⊑ H R ⟹ G(U,V) G(𝟙,H) h ⊑ G(𝟙,H) h R Proposition 9.4

thinning the parts by **U** in the first argument and by **V** in the second (**G(U,V)**), then solving by **H** and assembling by **h**, gives only what solving and assembling and then improving by **R** gives (**G(𝟙,H)hR**)



1	G(U,V) G(𝟙,H) h	
2	= G(U,V H) h	V H ⊑ H R
3	⊑ G(U,H R) h	
4	= G(𝟙,H) G(U,R) h	G(U,R) h ⊑ h R
5	⊑ G(𝟙,H) h R	

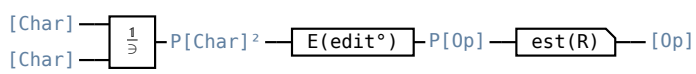
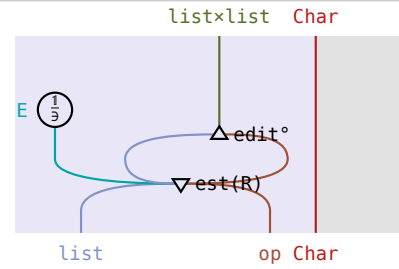
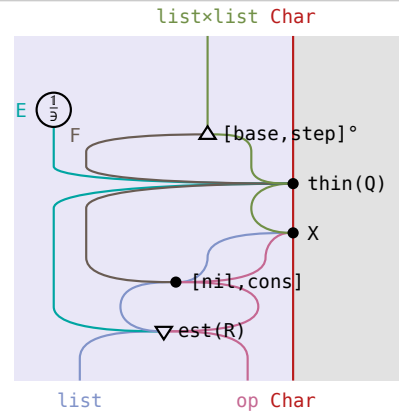
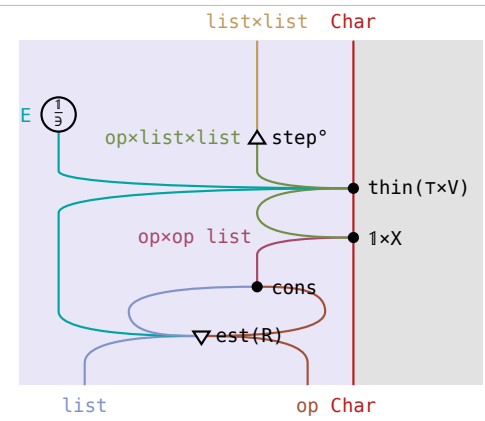
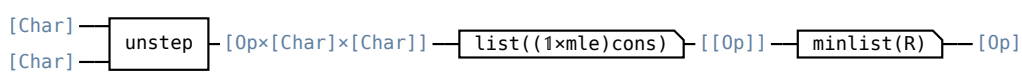
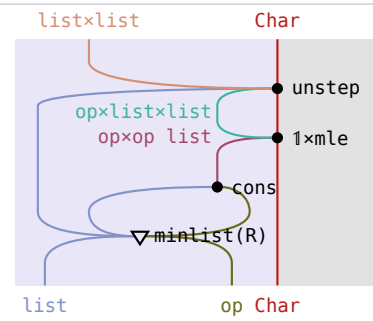
§9.2. The string edit problem

(9.2a)

	definition	type	meaning
1	(∃a. o = cpy(a)) ∨ (∃a. o = del(a)) ∨ (∃a. o = ins(a))	Type	an edit operation copies, deletes or inserts one character
2	FX = 1 + (Op × X)		F(Op,X) is the empty sequence or one operation paired with an X
3	edit ≜ ⦇[base,step]⦈	[Op]→[Char]×[Char]	the two strings an edit sequence turns one into the other
4	length = ⦇[zero,π₂ succ]⦈		the number of operations in a sequence
5	R ≜ length ≤ length°	[Op]→[Op]	es is no longer than fs
6	V ≜ suffix° × suffix°	[Char]×[Char]→ [Char]×[Char]	each string of the first pair is a suffix of the matching string of the second
7	l(val) Q l(val') ⟺ true, r(p) Q r(q) ⟺ V(π₂(p),π₂(q)), (∀val. ∀val'. u = l(val) ⟹ v = l(val') ⟹ false) ∧ (∀p. ∀q. u = r(p) ⟹ v = r(q) ⟹ false) ⟹ (u Q v ⟺ false) , Q(l(u),l(u)) , Q(r(p),r(q)) ⟺ V(π₂(p),π₂(q))	F([Char]×[Char])→ F([Char]×[Char])	the identity on the base summand; on the step summand any two operations, and the two pairs of strings by V
8	empty(p) ≜ p = ([],[])	[Char]×[Char]→ [Char]×[Char]	holds only at the pair of two empty strings
9	q ∈ unstep(p) ⟹ step(q) = p , unstep([],[]) = [] , unstep([a]⧺xs,[]) = [(del(a),(xs,[]))] , unstep([],[b]⧺ys) = [(ins(b),([],ys))] , unstep([a]⧺xs,[b]⧺ys) = if a = b then [(cpy(a),(xs,ys))] else [(del(a),(xs,[b]⧺ys)), (ins(b),([a]⧺xs,ys))]	[Char]×[Char]→ [Op×[Char]×[Char]]	every split of a pair of strings into one operation and a smaller pair; step puts it back
10	[nil,cons](l(u)) = [] , [nil,cons](r(o,es)) = [o]⧺es	Unit + Op × [Op]→[Op]	[nil,cons]: the empty edit sequence, or one operation in front of a sequence
11	base = ([],[]) , step(cpy(a),(xs,ys)) = ([a]⧺xs,[a]⧺ys) , step(del(a),(xs,ys)) = ([a]⧺xs,ys) , step(ins(a),(xs,ys)) = (xs,[a]⧺ys)	Unit + Op × [Char]×[Char]→ [Char]×[Char]	[base,step]: the two empty strings, or one operation added to the front of the pair

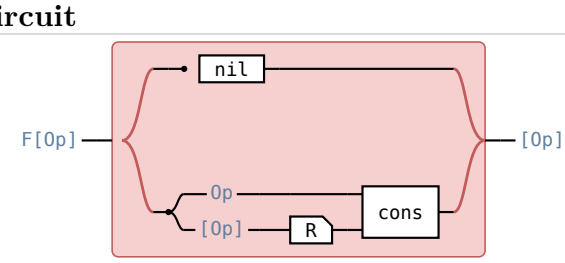
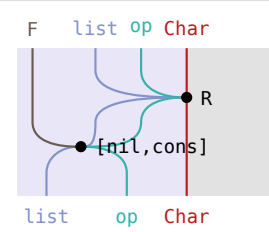
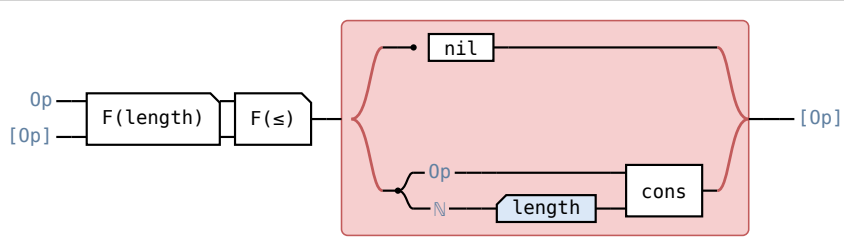
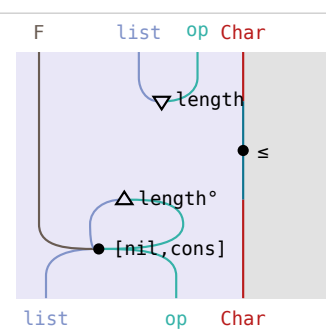
mle=(empty→nil,unstep list((𝟙×mle)cons) minlist(R)), (μX : $\frac{\texttt{[base,step]°}}{\ni}$ thin(Q) P(F(X) [nil,cons]) est(R)) ⊑ $\frac{\texttt{edit°}}{\ni}$ est(R) (9.2b)

a shortest edit sequence from which both strings can be reconstituted is one pass over the two of them, each step copying, deleting or inserting one character and the best sequence for what is left taken from the entries already computed

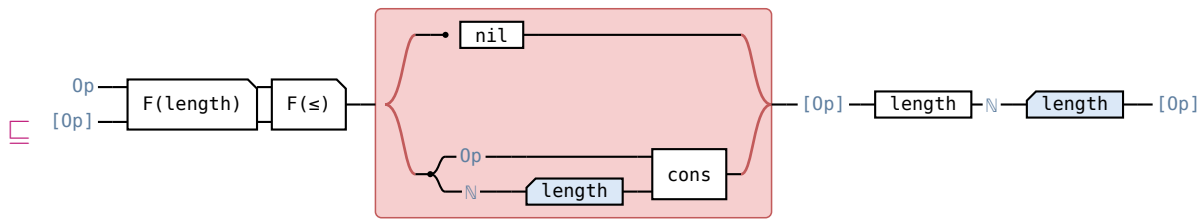
circuit	Hinze–Marsden
 the specification — `edit([]) = ([],[]), edit([op]+es) = step(op,edit(es))` , `edit ≜ ⦇[base,step]⦈`	
⊑ at `Q≜𝟙+(U×V)`. Monotonicity `F(R)α⊑αR` is Proposition 9.2 at `length≜⦇[zero,π₂ succ]⦈` with `succ` monotonic on ≤, so `cons` is monotonic on `R`. The thinning condition `QF(𝟙,edit°)α⊑F(𝟙,edit°)αR` over `F(Op,[Char]×[Char])` is Proposition 9.4 at `U≜⊤` — `F(⊤,R)α⊑αR`, left as an exercise, so any two operations may be compared — and `V≜suffix°×suffix°` , — `edit (𝟙×suffix)⊑R° edit` , `edit (suffix×𝟙)⊑R° edit` . `suffix=tail*` and `BA⊑CB⟹BA*⊑C*B` cut those to one step each: drop the operation that produced the head, or weaken its `cpy` to a `del`, never lengthening the sequence	
= Proposition 9.1: `base` and `step` have disjoint ranges, so the fixed point splits into one branch per summand — `empty`, the coreflexive on `(xs,ys)` with both lists empty, is where `base` returns	
⊑  `unstep` implements $\frac{\texttt{step°}}{\ni}$ `thin(U×V)` — at most two decompositions survive, a copy beating a delete or an insert wherever it is available — and `minlist(R)` implements `est(R)`. The same subproblem is solved many times over, so the running time is exponential in the two lengths	

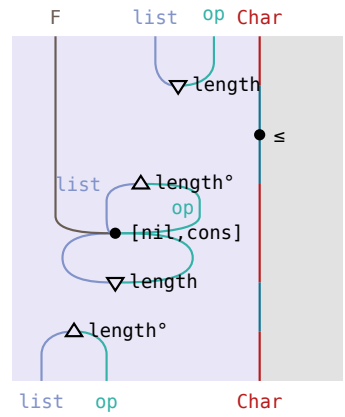
F(R) [nil,cons] ⊑ [nil,cons] R (9.2c)

comparing the tails by **R** and then putting an operation in front (**F(R)α**) relates only sequences that **R** also relates once the operation is in front (**αR**)

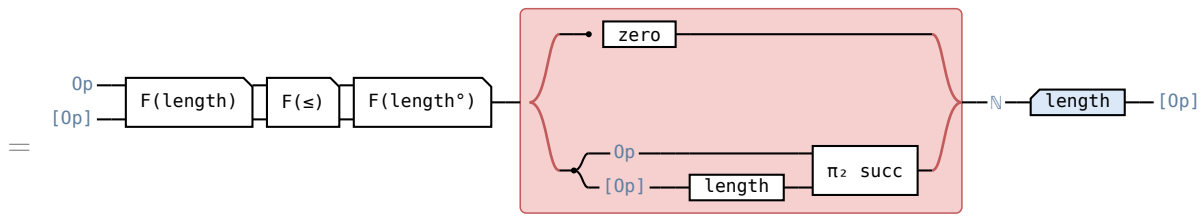
circuit	Hinze–Marsden
	
=  `R≜length≤length°` — `es R fs ⟺ length(es) ≤ length(fs)` , `R ≜ length ≤ length°` ; `F` preserves composition	

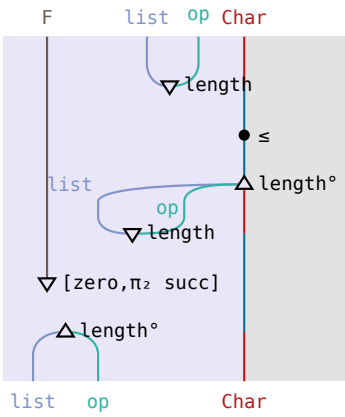
circuit	Hinze–Marsden



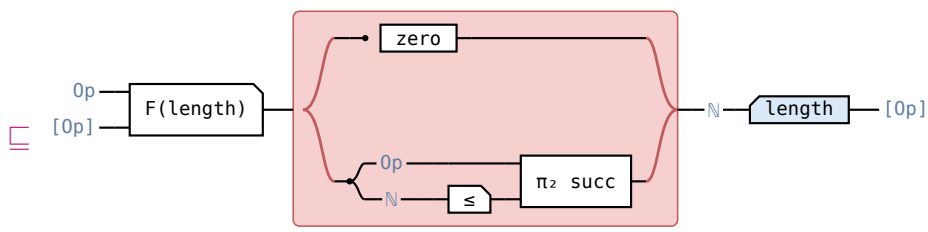


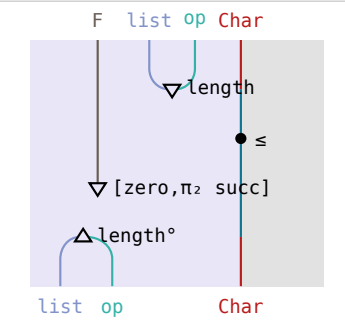
length is a map, so entire: $\mathbb{1} \sqsubseteq$ length length°



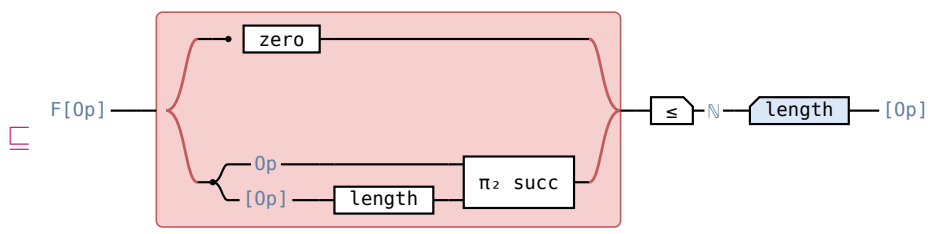


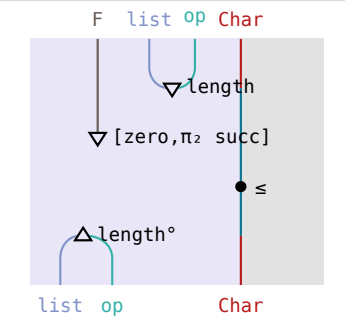
α length=F(length)[zero,π₂ succ]: length≜⦇[zero,π₂ succ]⦈ — es R fs ⟺ length(es) ≤ length(fs) , R ≜ length ≤ length°



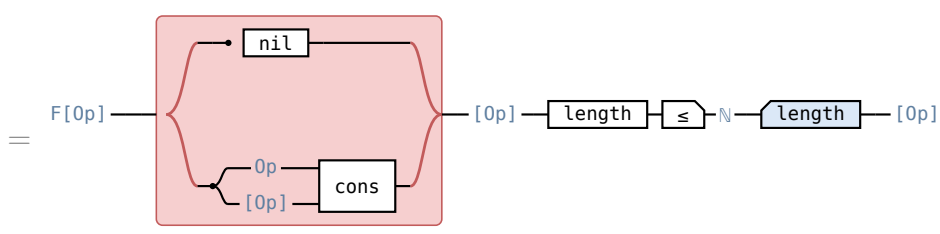


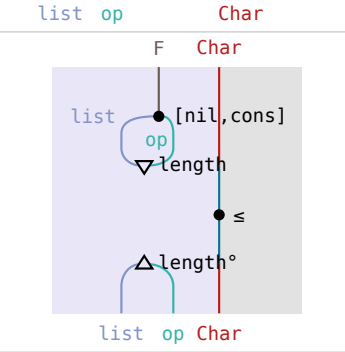
length is a map, so simple, and F is monotonic: F(length°)F(length)⊑F(𝟙)=𝟙



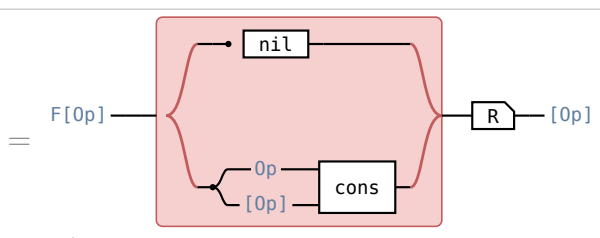


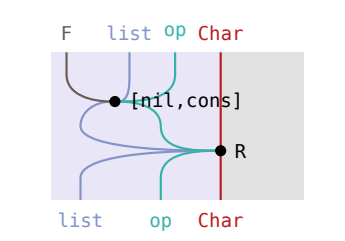
succ is monotonic on ≤: F(≤)[zero,π₂ succ]⊑[zero,π₂ succ]≤





F(length)[zero,π₂ succ]=α length again

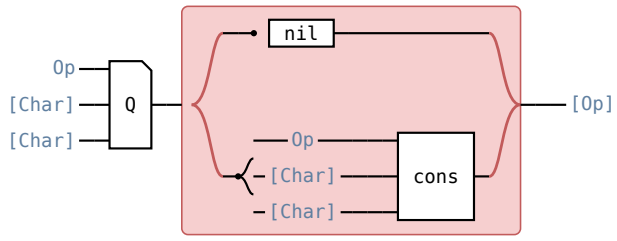
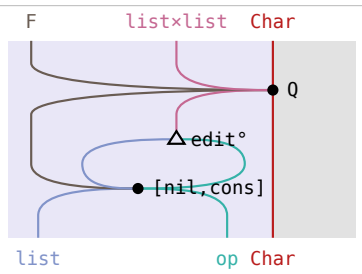
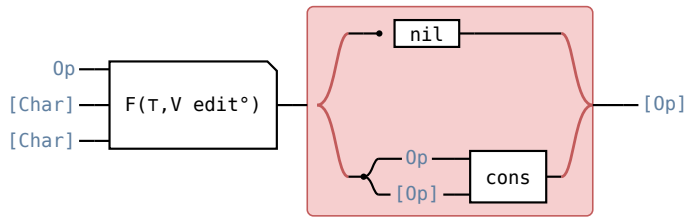
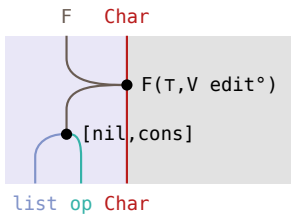
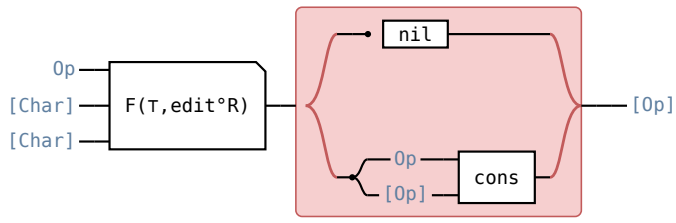
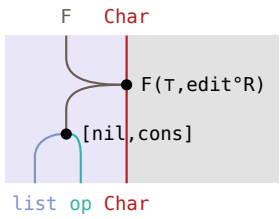
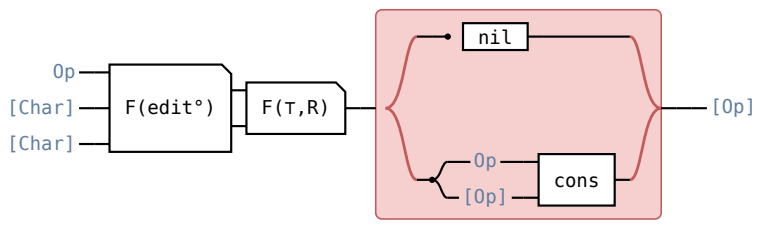
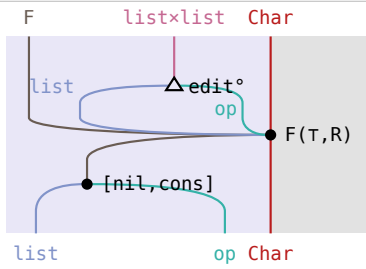
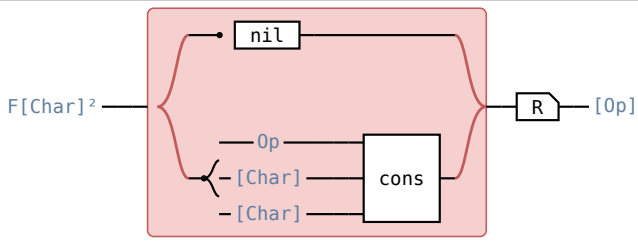
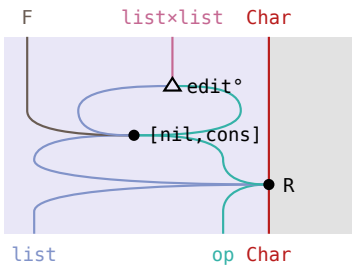




R≜length≤length° — es R fs ⟺ length(es) ≤ length(fs) , R ≜ length ≤ length°

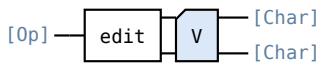
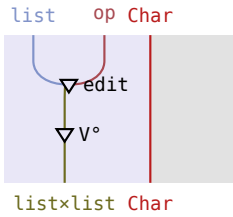
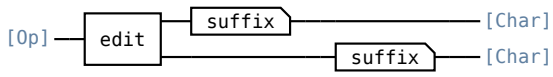
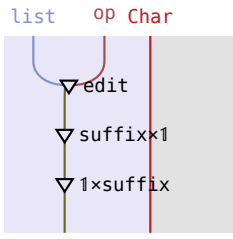
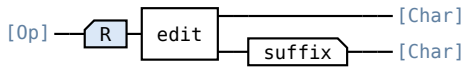
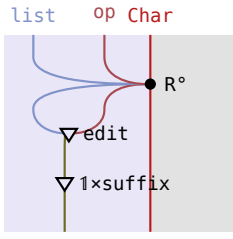
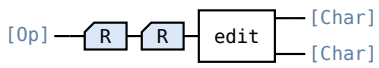
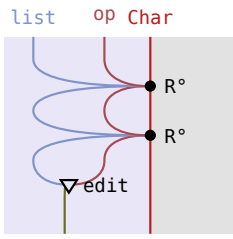
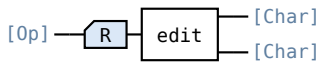
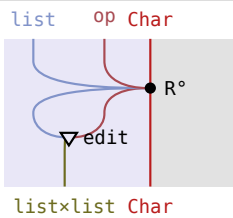
Q F(edit°) [nil,cons] ⊑ F(edit°) [nil,cons] R (9.2d)

every edit sequence built from a decomposition **Q** puts above a given one — any operation, each string lengthened at the front — is at least as long as one built from the given one

circuit	Hinze–Marsden
	
=  Q≜𝟙+(U×V) is F(U,V) at U≜⊤ — l(val) Q l(val') ⟺ true, r(p) Q r(q) ⟺ V(π₂(p),π₂(q)), (∀val. ∀val'. u = l(val) ⟹ v = l(val') ⟹ false) ∧ (∀p. ∀q. u = r(p) ⟹ v = r(q) ⟹ false) ⟹ (u Q v ⟺ false) , Q(l(u),l(u)) , Q(r(p),r(q)) ⟺ V(π₂(p),π₂(q)) ; and the bifunctor F preserves composition: F(U,V)F(𝟙,edit°)=F(U,V edit°)	
⊑  Proposition 9.4's second condition V edit°⊑edit° R — edit V° ⊑ R° edit	
=  F preserves composition: F(U,edit° R)=F(𝟙,edit°)F(U,R)	
⊑  Proposition 9.4's first condition F(⊤,R)α⊑αR, left as an exercise in the book: cons adds one to both lengths whatever the two operations are	

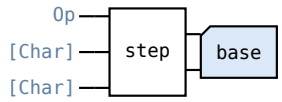
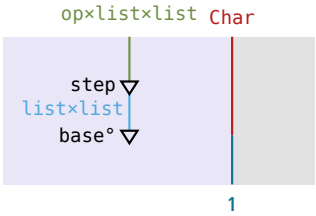
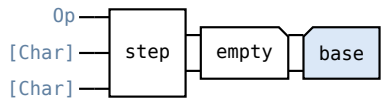
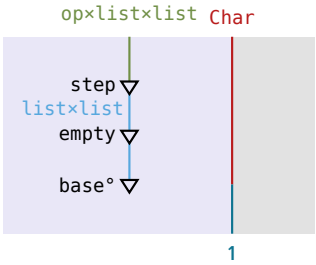
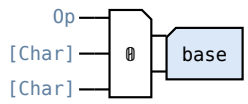
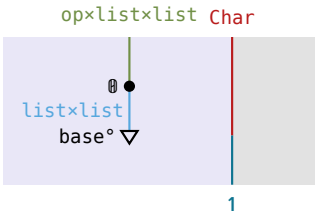
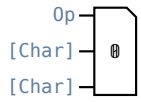
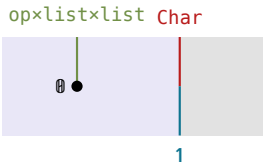
`edit V° ⊑ R° edit` (9.2e)

cutting a front off either string an edit sequence produces (`edit V°`) leaves a pair that a sequence no longer produces (`R° edit`)

circuit	Hinze–Marsden
	
=  `V≜suffix°×suffix°`, and `×` preserves composition	
⊑  `edit (suffix×𝟙)⊑R° edit`: drop the operation that produced the head, or weaken its `cpy` to an `ins`. Proved by induction at `suffix` directly, not through `suffix=tail*` and `BA⊑CB⟹BA*⊑C*B` as the book does	
⊑  `edit (𝟙×suffix)⊑R° edit`, the mirror image: `cpy` weakens to a `del`	
⊑  `R°` is transitive, `≤` being so	

`step base° = 𝟘` (9.2f)

no pair of strings is both a result of `step` and the result `([],[])` of `base`

circuit	Hinze–Marsden
	
=  `base` returns only `([],[])`, so `base°=empty base°`	
=  `step empty=𝟘`: `cpy` and `del` put a character on the left string, `ins` one on the right	
=  𝟘 composed with anything is 𝟘	

§9.2.1. Tabulation

(9.2.1a)

1
```
mle(xs,ys) = head(column(xs,ys)) , column(xs,ys)=[mle(u,ys)|u←tails(xs)]
```
= `column(xs)=⦇[fstcol(xs),nextcol(xs)]⦈, fstcol=list(del) tails`

the tabulation: `mle(xs,ys)` needs `mle(u,v)` for every tail `u` of `xs` and `v` of `ys`, so the columns are built right to left

2
```
column(xs,[b]⧺ys) = nextcol(xs,(b,column(xs,ys)))
```
= `nextcol(xs)(b,us)=⦇[base(b,last(us)),step(b)]⦈(xus), xus=zip(xs,zip(init(us),tail(us)))`

each column is a fold built bottom to top, over `xs` zipped with the adjacent pairs of the column to its right

3
```
base(b,u)=[[ins(b)]⧺u]
```
= `step(b)((a,(u,v)),ws)=(a=b→[[cpy(a)]⧺v]⧺ws, [bmin(R)([del(a)]⧺w,[ins(b)]⧺u)]⧺ws)`

`w=head(ws)`; these `base`, `step` are not `edit`'s. An entry depends on the one below it (a delete), the one to its right (an insert), and the one below that (a copy) — quadratic in the two lengths

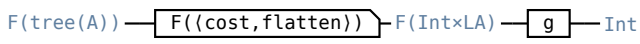
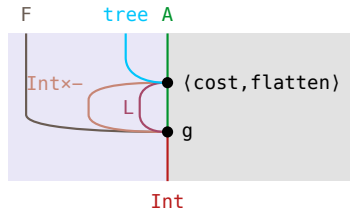
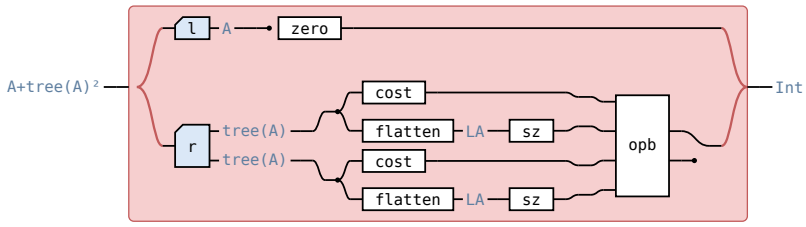
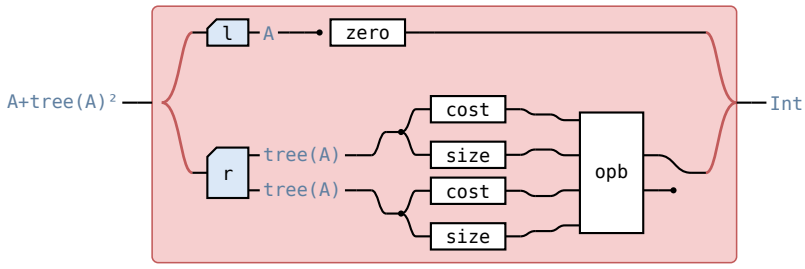
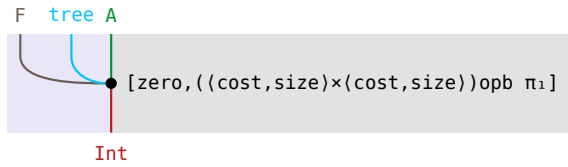
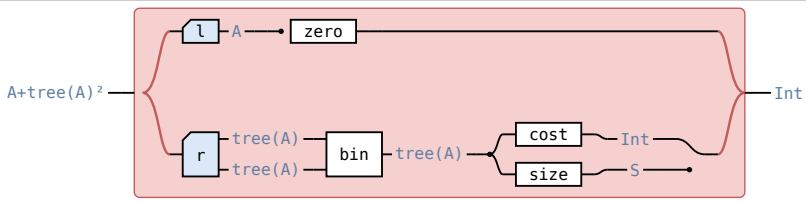
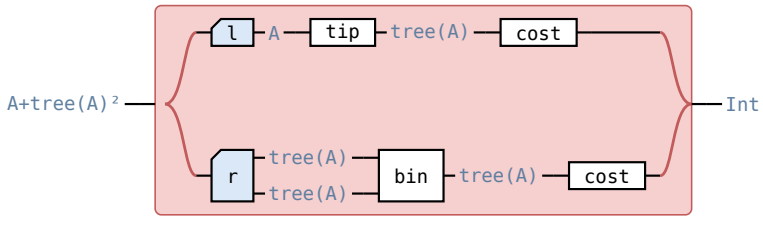
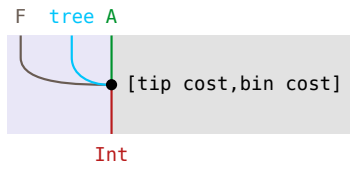
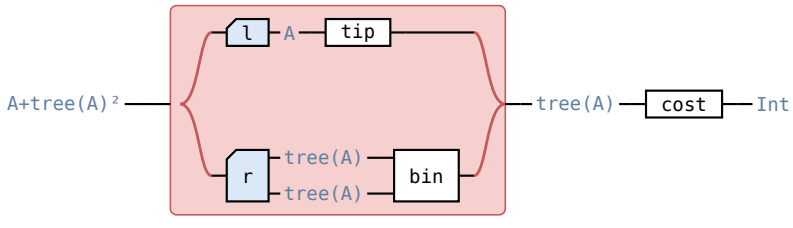
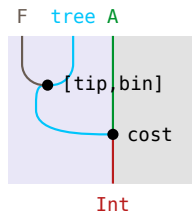
§9.3. Optimal bracketing

(9.3a)

	definition	type	meaning
1	tree(A) ≜ tip(A) | bin(tree(A),tree(A))	Type	a tree is a tip holding one value, or a node joining two trees; h≜[tip,bin]
2	⦇[wrap,cat]⦈ = flatten	tree(A)→LA	the tips of a tree, left to right; H=flatten° builds every tree over a list
3	opb(p) ≜ ((cb(π₂(π₁(p)),π₂(π₂(p))) + π₁(π₁(p))) + π₁(π₂(p)), sb(π₂(π₁(p)),π₂(π₂(p))))	Int×S	joining two subtrees costs cb of their sizes plus the cost of each
4	⟨cost,size⟩(tip(a)) = (0,st(a)), ⟨cost,size⟩(bin(xs,ys)) = opb(⟨cost,size⟩(xs),⟨cost,size⟩(ys))	tree(A)→Int×S	the cost of evaluating a bracketing, paired with the size of its result
5	sz([a]) = st(a), sz([a]+xs) = sb(st(a),sz(xs)) , Assoc(sb) ⟹ π₂(⟨cost,size⟩(xs)) = sz(flatten(xs))	LA→S	sb associative, so the size depends only on the flattened list
6	R ≜ cost ≤ cost°	tree(A)→tree(A)	t costs no more than t'
7	g ≜ [⊸ 0,(𝟙×sz)² opb π₁]	F(Int×LA)→Int	the cost of the top step alone: zero at a tip, cb of the two sizes at a node
8	nonsingle(xs) ≜ ∃a. ∃zs. xs = [a]+zs	LA→LA	holds at the lists of two or more elements
9	splits ≜ ⟨inits⁺,tails⁺⟩ zip	LA→[L A×L A]	every way to cut a list into two non-empty parts — an implementation of $\frac{\texttt{cat}^\circ}{\ni}$
10	array ≜ inits list(row)	LA→[[tree A]]	the best trees of every segment, held as rows
11	row ≜ tails list(mct)	LA→[tree A]	the best tree of every non-empty suffix
12	col ≜ inits list(mct)	LA→[tree A]	the best tree of every non-empty prefix
13	mix ≜ zip list(bin) minlist(R)	[tree A]×[tree A]→ tree(A)	join the matching prefix and suffix trees and keep a cheapest
14	next ≜ ⟨π₁,mix⟩ snoc	[tree A]×[tree A]→ [tree A]	add the next best tree to the end of the column
15	process ≜ ((tip wrap) × 𝟙) loop(next)	A×[[tree A]]→ [tree A]	start the column at a tip and run next over the rows

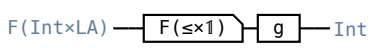
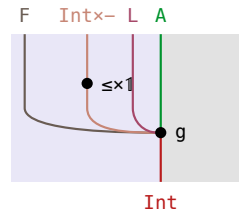
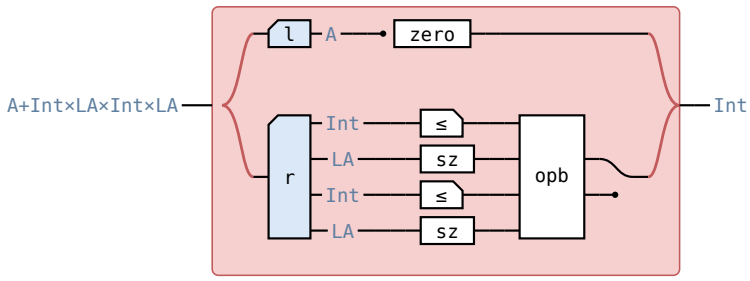
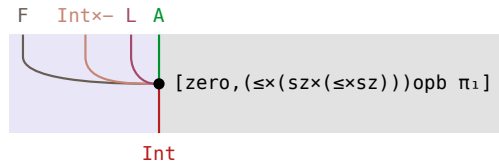
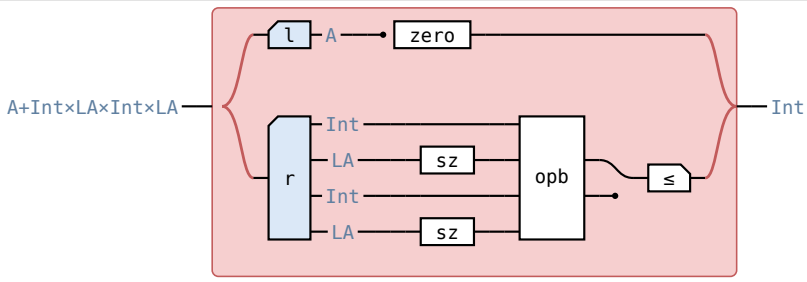
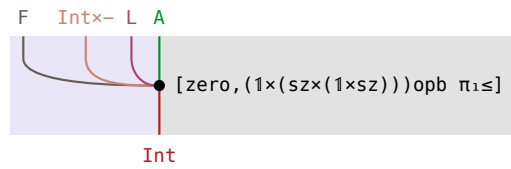
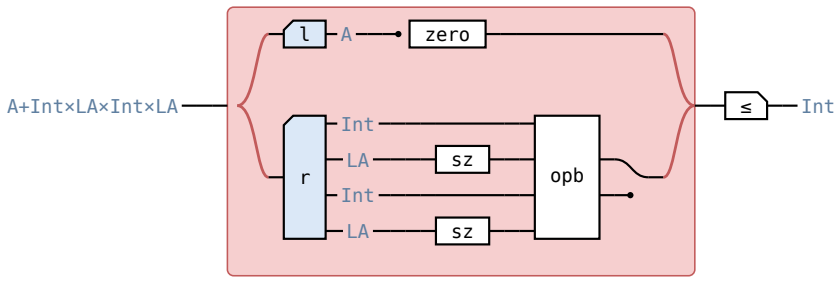
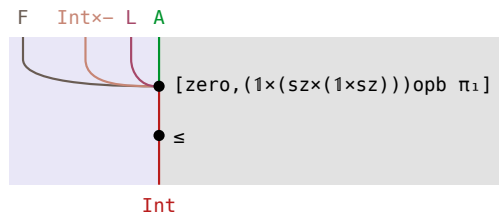
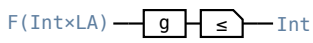
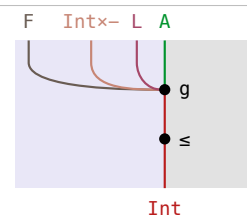
Assoc(sb) ⟹ F(⟨cost,flatten⟩) g = [tip,bin] cost (9.5)

pairing each subtree with its cost and its flattening and then applying **g** gives what building the tree by **[tip,bin]** and taking its **cost** gives

circuit	Hinze–Marsden
	
=  definition of g; coproducts and products — g ≜ [⊸ 0,(𝟙×sz)² opb π₁]	
=  flatten sz=size, since sb is associative — sz([a]) = st(a), sz([a]+xs) = sb(st(a),sz(xs)) , Assoc(sb) ⟹ π₂(⟨cost,size⟩(xs)) = sz(flatten(xs))	
=  ⟨cost,size⟩≜⦇[opt,opb]⦈, at a node bin⟨cost,size⟩ =⟨cost,size⟩² opb	
=  tip cost=zero; products, ⟨cost,size⟩π₁=cost	
=  coproducts, [tip cost,bin cost]=[tip,bin] cost	

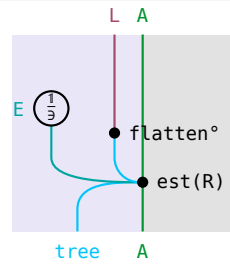
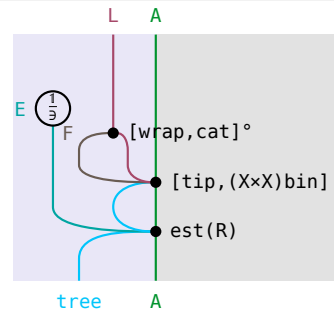
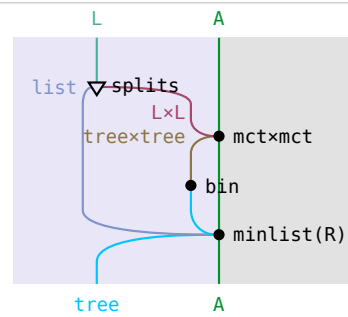
F(≤ × 𝟙) g ⊑ g ≤ (9.6)

raising the cost of either subtree (≤×𝟙 in both slots) and then applying **g** returns a value at least what **g** returns on the costs before raising (**g≤**)

circuit	Hinze–Marsden
	
=  definition of g — g ≜ [⊸ 0,(𝟙×sz)² opb π₁]	
⊑  definition of opb, and + monotonic	
⊑  ≤ reflexive, so zero⊑zero ≤; coproducts	
=  definition of g — g ≜ [⊸ 0,(𝟙×sz)² opb π₁]	

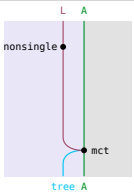
mct=(single→head tip,⟨init col,tail row⟩ mix), Assoc(sb) ⟹ (μX : $\frac{\text{[wrap,cat]°}}{\ni}$ P([tip,(X × X) bin]) est(R)) ⊑ $\frac{\text{flatten°}}{\ni}$ est(R) (9.3b)

a least-cost bracketing of $a_1 \oplus \cdots \oplus a_n$ is read off an array holding one best tree per non-empty segment, each entry built from the column to its left and the row below it

circuit	Hinze–Marsden
LA — $\frac{1}{\ni}$ — P(LA) — E(flatten°) — P(tree(A)) — est(R) — tree(A) the specification — flatten(tip(a)) = [a], flatten(bin(xs,ys)) = cat(flatten(xs),flatten(ys)) , ⦇[wrap,cat]⦈ = flatten	
LA — $\frac{1}{\ni}$ — P(LA) — E([wrap,cat]°) — PF(LA) — P([tip,(X×X)bin]) — P(tree(A)) — est(R) — tree(A) split the list in every way, bracket both halves, join. The condition is monotonicity **in context**, ⊑ F(R∩(flatten flatten°))h⊑hR — only trees with the same flattening are compared — which is Proposition 9.3 at H°=flatten, a map, with (9.5) [tip,bin] cost=(𝟙+⟨cost,flatten⟩²)g (the cost of a node reads only the cost and the flattening of its two subtrees) and (9.6) (𝟙+(≤×𝟙)²)g⊑g≤ (g monotonic on ≤ in its two cost arguments)	
= LA — $\frac{1}{\ni}$ — P(LA) — E(cat°) — P(LA²) — P((X×X)bin) — P(tree(A)) — est(R) — tree(A) Proposition 9.1: wrap and cat have disjoint ranges, and single is the coreflexive on singleton lists, where wrap returns	
LA — splits — [LA²] — list((mct×mct)bin) — [tree(A)] — minlist(R) — tree(A) ⊑ splits≜⟨inits⁺,tails⁺⟩ zip implements $\frac{\text{cat°}}{\ni}$ and minlist(R) implements est(R). Exponential, since the segments of one list overlap	
= mct=(single→head tip,⟨init col,tail row⟩ mix) (9.7) the tabulation: mct xs is needed for every non-empty segment xs, so the values are held as an array of rows, array≜inits list(row), row≜tails list(mct), col≜inits list(mct), mix≜zip list(bin) minlist(R)	
= col=(single→head tip wrap,⟨init col,tail row⟩ next) (9.8) cons col=(𝟙×array) process (9.9) row=(single→head tip wrap,⟨mct,tail row⟩ cons) (9.10) a column extends the column to its left, a row the row below it; (9.9) is (9.8) rewritten as a loop, next≜⟨π₁,mix⟩ snoc, process≜((tip wrap)×𝟙) loop(next)	
= array=⦇[fstcol,addcol]⦈, fstcol≜tip wrap wrap addcol≜⟨π₁ tip wrap,step⟩ cons, step≜⟨process tail,π₂⟩ zip list(cons) the program: one fold building the array column by column, cubic in the length of the input	

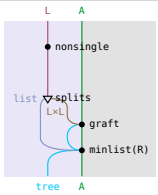
nonsingle mct = nonsingle ⟨init col,tail row⟩ mix (9.7)

on a list of two or more elements, **mct** gives what **mix** gives on the column of its **init** and the row of its **tail**



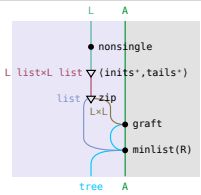
(1)

=



(2)

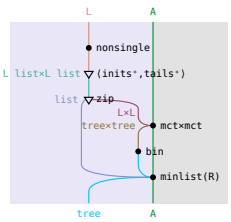
=



(3)

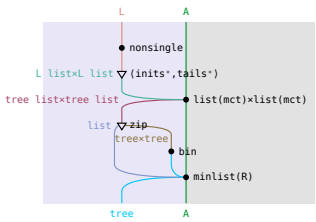
1	nonsingle mct	nonsingle mct = nonsingle splits list(graft) minlist(R)
2	= nonsingle splits list(graft) minlist(R)	splits = ⟨inits⁺,tails⁺⟩ zip
3	= nonsingle ⟨inits⁺,tails⁺⟩ zip list(graft) minlist(R)	graft ≜ (mct × mct) bin

=



(4)

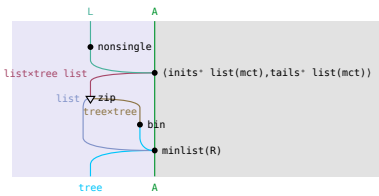
=



(5)

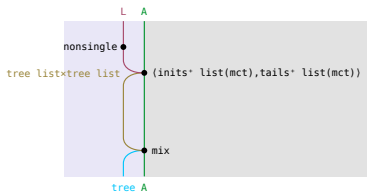
4	= nonsingle ⟨inits⁺,tails⁺⟩ zip list(mct × mct) list(bin) minlist(R)	zip list(f × g) M = (list(f) × list(g)) zip M
5	= nonsingle ⟨inits⁺,tails⁺⟩ (list(mct) × list(mct)) zip list(bin) minlist(R)	⟨R,S⟩ (U × V) = ⟨R U,S V⟩

=



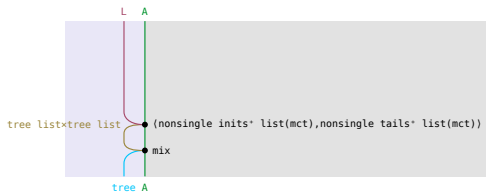
(6)

=



(7)

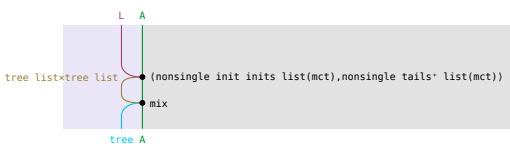
=



(8)

6	= nonsingle ⟨inits⁺ list(mct),tails⁺ list(mct)⟩ zip list(bin) minlist(R)	mix = zip list(bin) minlist(R)
7	= nonsingle ⟨inits⁺ list(mct),tails⁺ list(mct)⟩ mix	nonsingle ⟨U,V⟩ = ⟨nonsingle U,nonsingle V⟩
8	= ⟨nonsingle inits⁺ list(mct),nonsingle tails⁺ list(mct)⟩ mix	nonsingle inits⁺ M = nonsingle init inits M

=

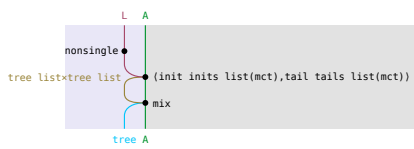


(9)

=

(10)

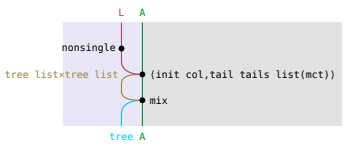
=



(11)

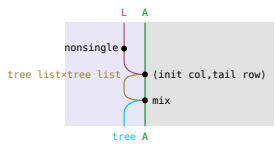
9	= ⟨nonsingle init inits list(mct),nonsingle tails⁺ list(mct)⟩ mix	nonsingle tails⁺ M = nonsingle tail tails M
10	= ⟨nonsingle init inits list(mct),nonsingle tail tails list(mct)⟩ mix	nonsingle ⟨U,V⟩ = ⟨nonsingle U,nonsingle V⟩
11	= nonsingle ⟨init inits list(mct),tail tails list(mct)⟩ mix	col = inits list(mct)

=



(12)

=

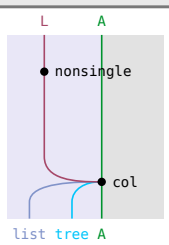


(13)

12	= nonsingle ⟨init col,tail tails list(mct)⟩ mix	row = tails list(mct)
13	= nonsingle ⟨init col,tail row⟩ mix	

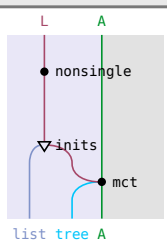
nonsingle col = nonsingle ⟨init col,tail row⟩ next (9.8)

on a list of two or more elements, the column is what **next** makes of the column of its **init** and the row of its **tail**



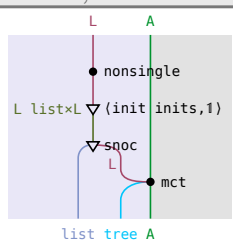
(1)

≜



(2)

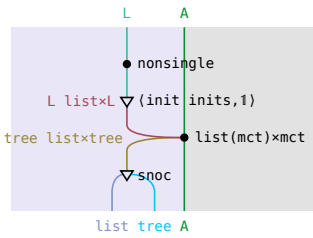
=



(3)

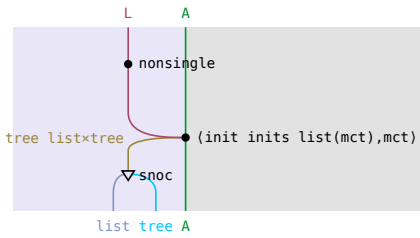
1	nonsingle col	col ≜ inits list(mct)
2	≜ nonsingle inits list(mct)	nonsingle inits M = nonsingle ⟨init inits,𝟙⟩ snoc M
3	= nonsingle ⟨init inits,𝟙⟩ snoc list(mct)	snoc list(f) = (list(f) × f) snoc

=



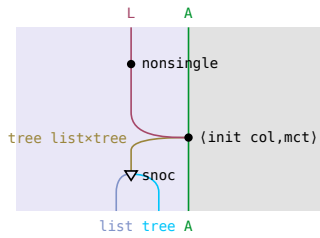
(4)

=



(5)

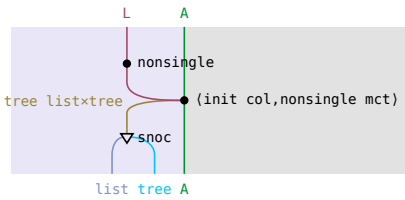
=



(6)

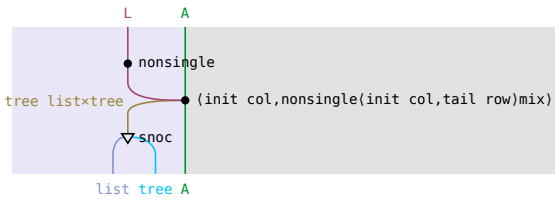
4	= nonsingle ⟨init inits,𝟙⟩ (list(mct) × mct) snoc	⟨R,S⟩ (U × V) = ⟨R U,S V⟩
5	= nonsingle ⟨init inits list(mct),mct⟩ snoc	col = inits list(mct)
6	= nonsingle ⟨init col,mct⟩ snoc	nonsingle ⟨U,V⟩ = nonsingle ⟨U,nonsingle V⟩

=



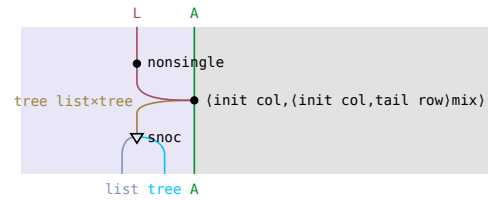
(7)

=



(8)

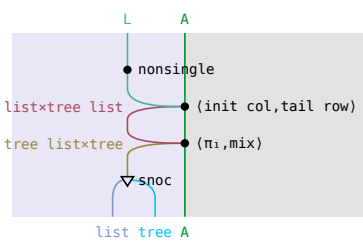
=



(9)

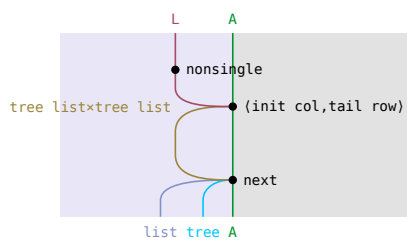
7	= nonsingle ⟨init col,nonsingle mct⟩ snoc	(9.7) nonsingle mct = nonsingle ⟨init col,tail row⟩ mix
8	= nonsingle ⟨init col,nonsingle ⟨init col,tail row⟩ mix⟩ snoc	nonsingle ⟨U,V⟩ = nonsingle ⟨U,nonsingle V⟩
9	= nonsingle ⟨init col,⟨init col,tail row⟩ mix⟩ snoc	⟨init col,V⟩ ⟨π₁,M⟩ = ⟨init col,⟨init col,V⟩ M⟩

=



(10)

=

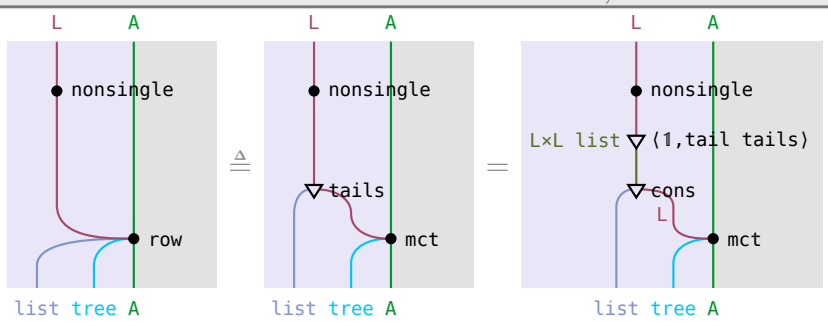


(11)

10	= nonsingle ⟨init col,tail row⟩ ⟨π₁,mix⟩ snoc	next = ⟨π₁,mix⟩ snoc
11	= nonsingle ⟨init col,tail row⟩ next	

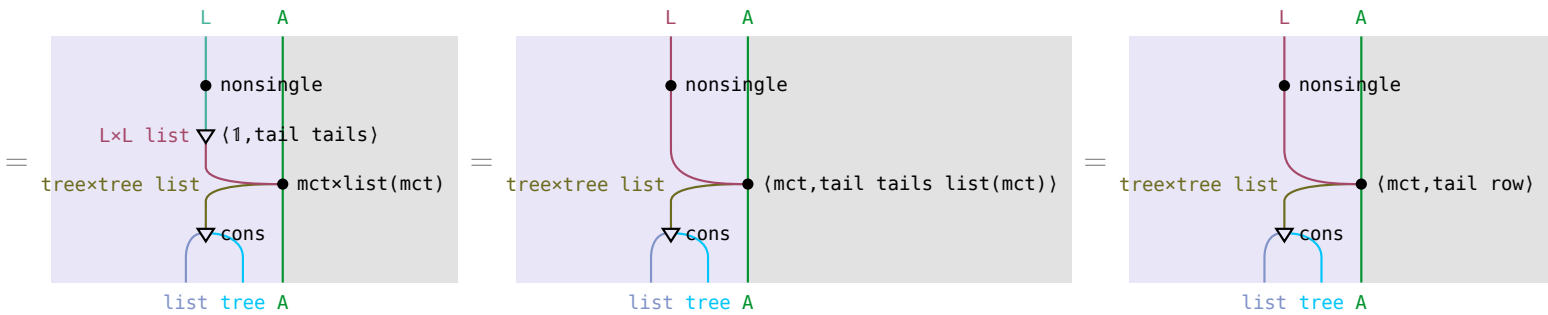
nonsingle row = nonsingle ⟨mct,tail row⟩ cons (9.10)

on a list of two or more elements, the row is **mct** of the whole list consed onto the row of its **tail**



(1) (2) (3)

1	nonsingle row	row ≜ tails list(mct)
2	≜ nonsingle tails list(mct)	nonsingle tails M = nonsingle ⟨𝟙,tail tails⟩ cons M
3	= nonsingle ⟨𝟙,tail tails⟩ cons list(mct)	cons list(f) = (f × list(f)) cons

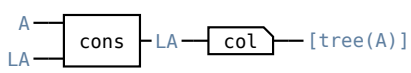
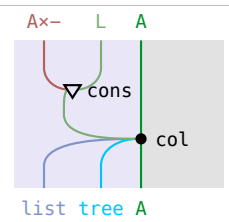
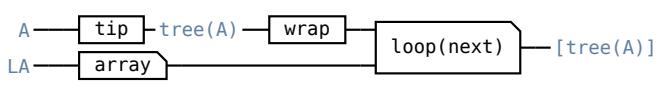
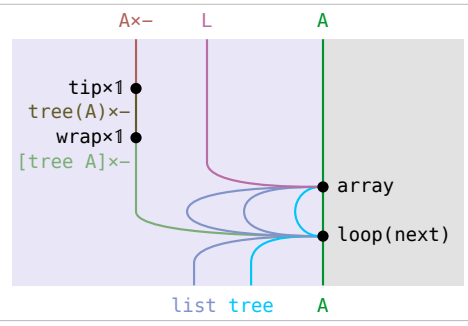
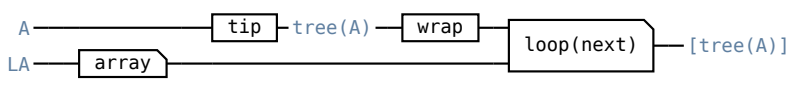
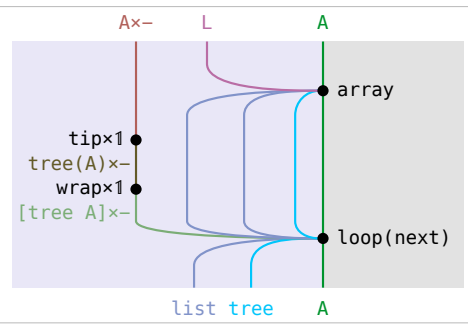
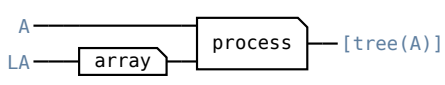
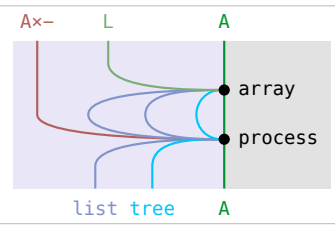


(4) (5) (6)

4	= nonsingle ⟨𝟙,tail tails⟩ (mct × list(mct)) cons	⟨R,S⟩ (U × V) = ⟨R U,S V⟩
5	= nonsingle ⟨mct,tail tails list(mct)⟩ cons	row = tails list(mct)
6	= nonsingle ⟨mct,tail row⟩ cons	

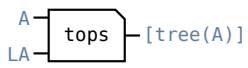
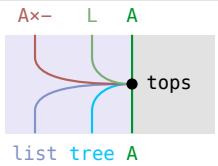
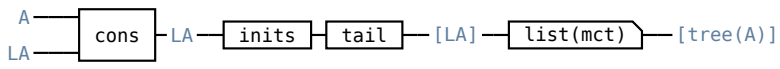
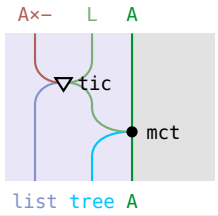
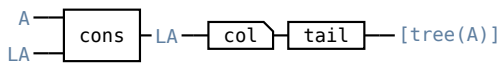
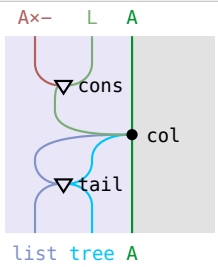
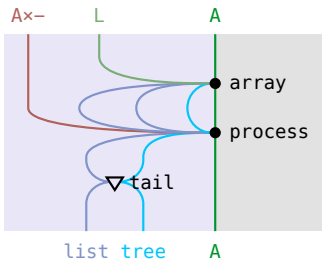
cons col = (𝟙 × array) process (9.9)

the column of **a** consed onto **x** is what **process** makes of **a** and the array of **x**

circuit	Hinze–Marsden
	
 = Exercise 9.13 with k≜cons col, g≜tip wrap, h≜row, f≜next: its two equations are (9.8) — (9.8) nonsingle col = nonsingle ⟨init col,tail row⟩ next — at a:[b] and at a:(u++[b]), with [a] col=[tip(a)]	
 = products	
 = definition of process	

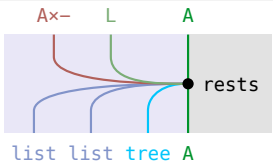
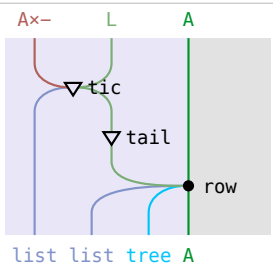
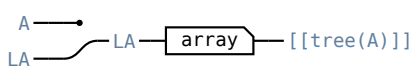
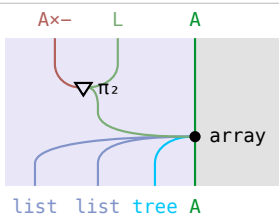
tops = (𝟙 × array) process tail (9.3c)

the trees at the top of the new rows are what **process** makes of **a** and the array of **x**, less its first

circuit	Hinze–Marsden
	
=  definition of `tops≜tic list(mct)`, our name	
=  `tail list(f)=list(f) tail`; definition of `col`	
= (9.9) — (9.9) `cons col = (𝟙 × array) process`	

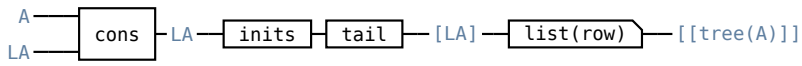
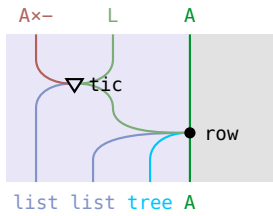
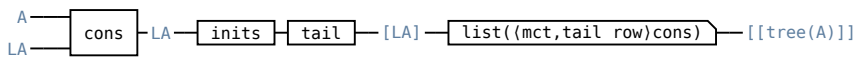
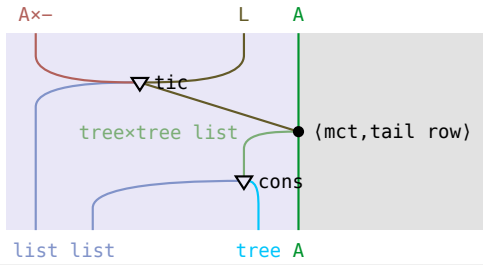
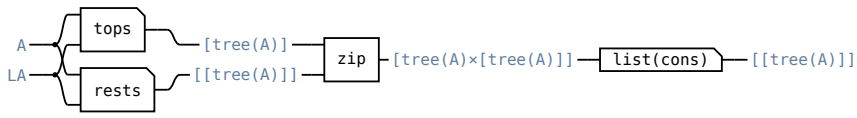
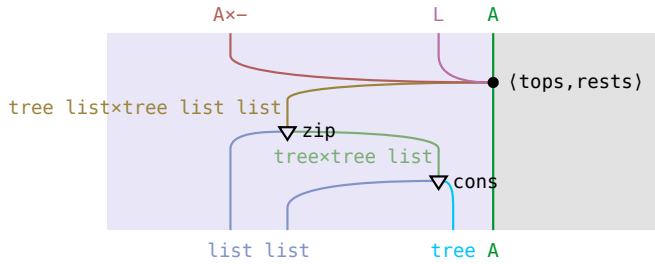
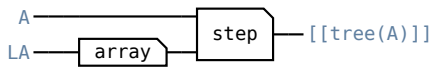
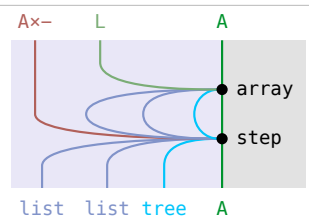
rests = π₂ array (9.3d)

the rest of each new row is the array of **x**

circuit	Hinze–Marsden
	
= definition of `rests≜tic list(tail row)`, our name	
=  `tic list(tail)=π₂ inits`; definition of `array`	

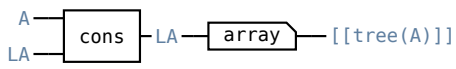
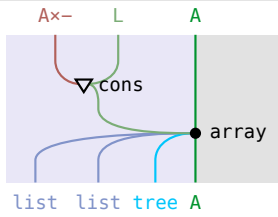
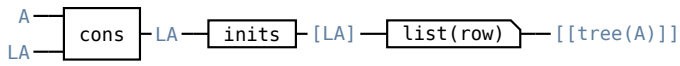
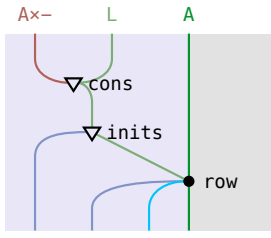
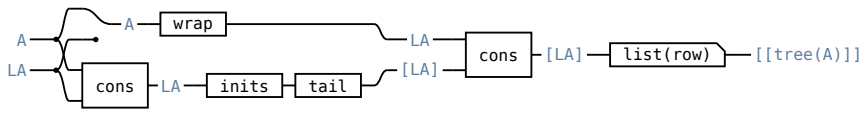
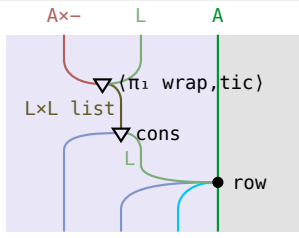
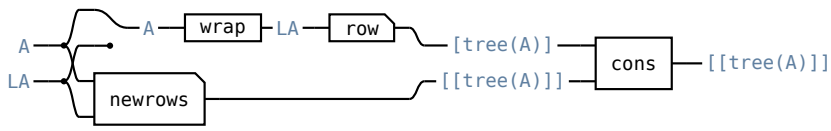
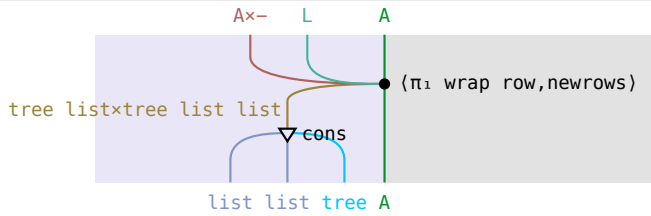
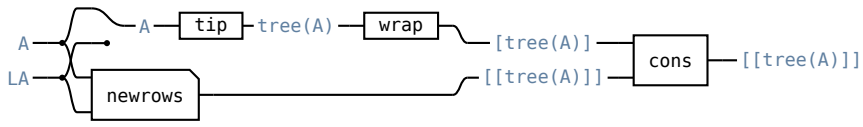
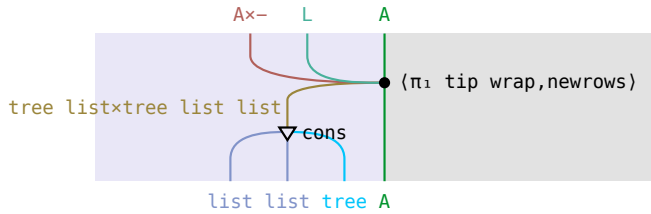
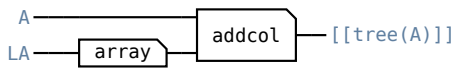
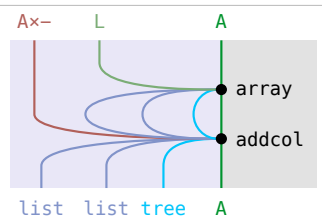
`newrows = (𝟙 × array) step` (9.3e)

the rows of the array of **`a:x`** below its first are what **`step`** makes of **`a`** and the array of **`x`**

circuit	Hinze–Marsden
=  definition of `newrows≜tic list(row)`, our name	
=  (9.10) — (9.10) `nonsingle row = nonsingle ⟨mct,tail row⟩` `cons` — on the non-singleton `tic` lists	
=  `list⟨f,g⟩=⟨list(f),list(g)⟩ zip`; products; definitions of `tops` and `rests`	
=  `tops = (𝟙 × array) process tail` and `rests = π₂ array`; products; definition of `step`	

`cons array = (𝟙 × array) addcol` (9.3f)

the array of **a** consed onto **x** is what **addcol** makes of **a** and the array of **x**

circuit	Hinze–Marsden
	
=  definition of `array`	
=  `cons inits=⟨π₁ wrap,tic⟩ cons`, abbreviating `cons inits tail` by `tic`	
=  `cons list(f)=(f×list(f)) cons`; definition of `newrows`	
=  `wrap row=tip wrap`	
=  `newrows = (𝟙 × array) step` ; products; definition of `addcol`	

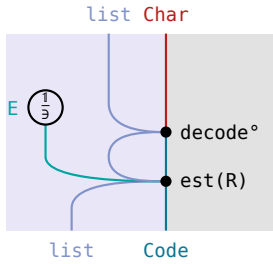
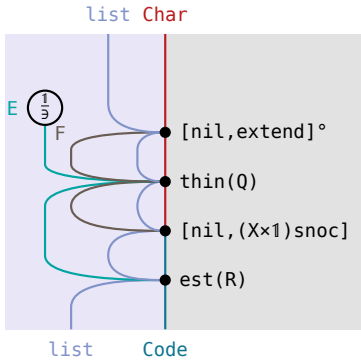
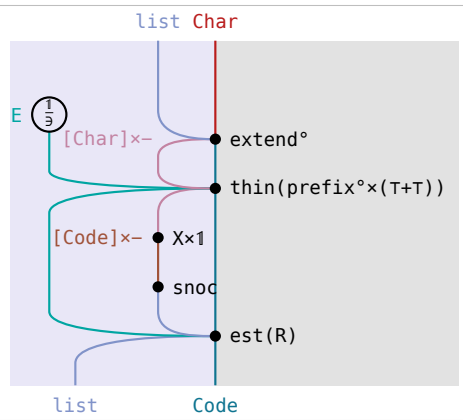
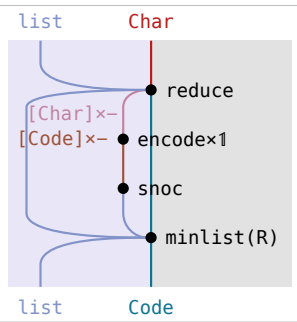
§9.4. Data compression

(9.4a)

	definition	type	meaning
1	`String ≜ [Char]`	`Type`	a string is a snoc-list of characters
2	`LA ≜ [A]`	`Type`	a non-empty snoc-list: one value, then any number more appended
3	`Code ≜ sym(Char) | ptr(String,String)`	`Type`	a code is one character, or a pointer back to a repeated piece
4	`extend((xs,sym(a)),x) = x = snoc(xs,a), extend((xs,ptr(ys,zs)),x) = x = cat(xs,zs) ∧ zs ≠ [] ∧ (init⁺)(cat(ys,zs),cat(xs,zs))`	`String×Code→[Char]`	a symbol appends its character; a pointer appends `zs` when `ys#zs` is a proper prefix of `xs#zs`
5	`decode ≜ ⦇[nil,extend]⦈`	`[Code]→[Char]`	rebuild the string from its codes, left to right; `H=decode°`
6	`bytes(sym(a)) = c, bytes(ptr(a,a')) = p`	`Code→ℕ`	a symbol costs `c`, a pointer costs `p`
7	`⦇[nil,distr [𝟙×c,𝟙×p] plus]⦈ = size`	`[Code]→ℕ`	the total cost of a code sequence
8	`R ≜ size ≤ size°`	`[Code]→[Code]`	`u` costs no more than `v`
9	`prefix ≜ cat° π₁`	`[Char]→[Char]`	split the string in two and keep the first part
10	`(T+T) ≜ [sym,ptr]° (T + T) [sym,ptr]`	`Code→Code`	any symbol to any symbol, any pointer to any pointer
11	`Q ≜ F((T+T),prefix°)`	`F[Char]→F[Char]`	compare two decompositions by the prefix order on the strings and `U` on the codes
12	`q ∈ reduce(ws) ⟺ extend(q,ws)`	`[Char]→[String×Code]`	the last code of a string: one symbol, or a pointer to its longest repeated tail

$$(\mu X : \frac{[\mathtt{nil},\mathtt{extend}]^\circ}{\ni}\ \mathtt{thin}(Q)\ P([\mathtt{nil},(X \times \mathbb{1})\ \mathtt{snoc}])\ \mathtt{est}(R)) \sqsubseteq \frac{\mathtt{decode}^\circ}{\ni}\ \mathtt{est}(R)\ , \qquad (9.4b)$$
$$\mathtt{encode}=(\mathtt{null}\to\mathtt{nil},\mathtt{reduce}\ \mathtt{list}((\mathtt{encode}\times\mathbb{1})\mathtt{snoc})\ \mathtt{minlist}(R))$$

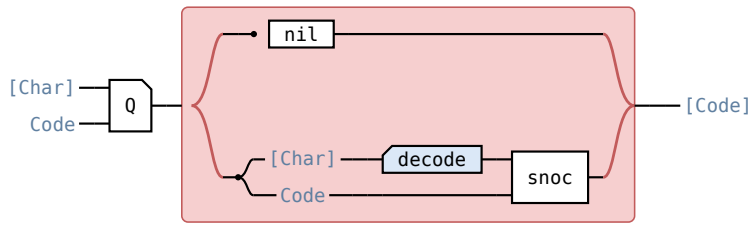
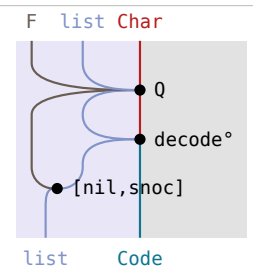
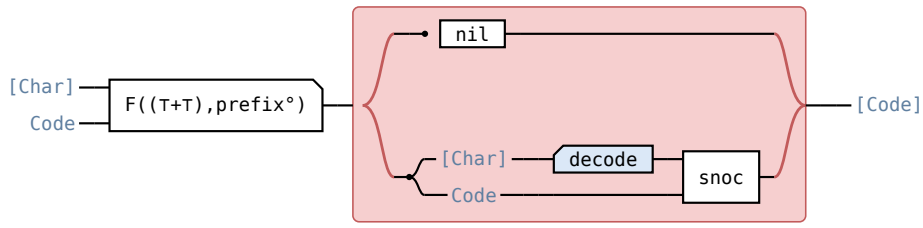
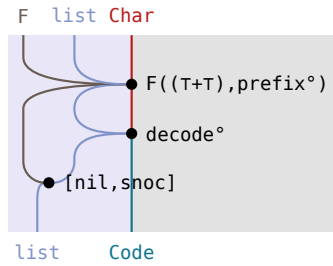
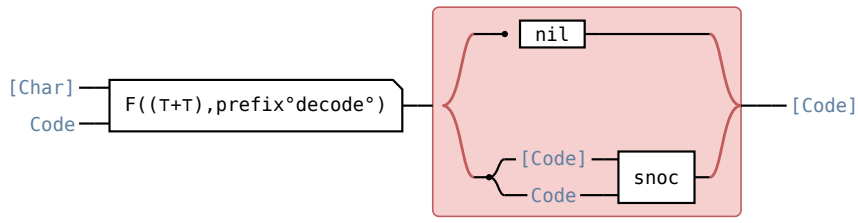
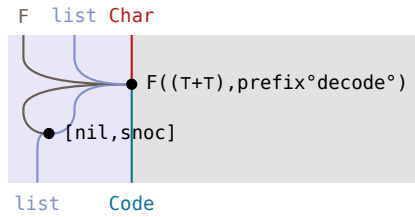
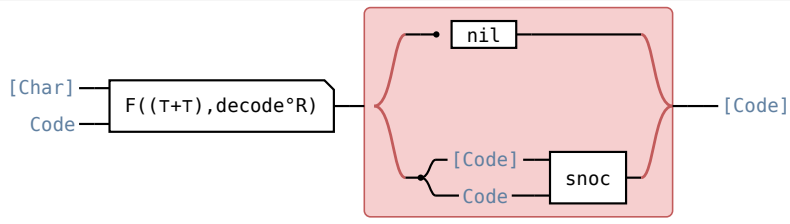
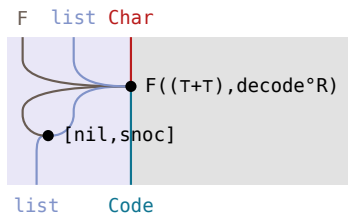
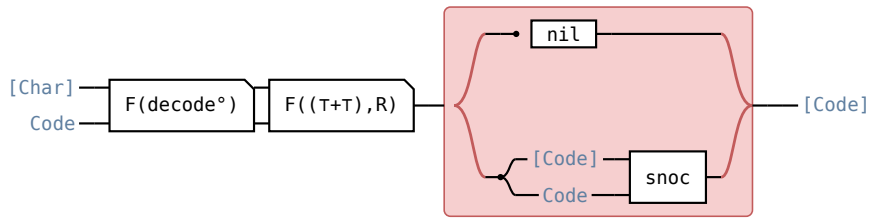
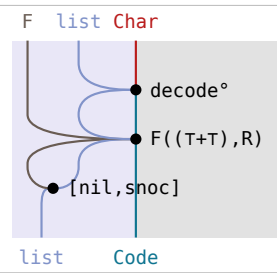
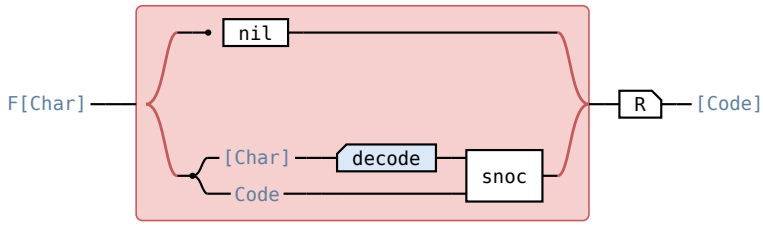
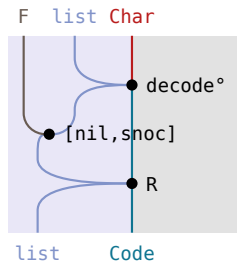
a smallest code sequence decoding to the given string is built from the right, each step emitting the last character as a symbol or ending with a pointer back into what has already been decoded

circuit	**Hinze–Marsden**
[Char] — $\frac{1}{\ni}$ — P[Char] — E(decode°) — P[Code] — est(R) — [Code] the specification — $\mathtt{decode} \triangleq (\!\![\mathtt{nil},\mathtt{extend}]\!\!)$	
[Char] — $\frac{1}{\ni}$ — P[Char] — E([nil,extend]°) — thin(Q) — PF[Char] — P([nil,(X×𝟙)snoc]) — P[Code] — est(R) — [Code] at $Q \triangleq F(T+T,\mathtt{prefix}^\circ)=\mathbb{1}+(\mathtt{prefix}^\circ\times(T+T))$, the two T on symbols and on pointers. Monotonicity $F(R)\alpha\sqsubseteq\alpha R$ is routine, the two costs being constants: $(T+T)[c,p]=[c,p]$. ⊒ Proposition 9.4 splits the thinning condition in two — $F(T+T,R)\alpha\sqsubseteq\alpha R$, and $\mathtt{decode}\ \mathtt{prefix}\sqsubseteq R^\circ\ \mathtt{decode}$, for which $\mathtt{decode}\ \mathtt{init}\sqsubseteq R^\circ\ \mathtt{decode}$ is enough: dropping the last character of the output shortens or removes the last code element and never raises the cost. So between a symbol and a pointer nothing can be decided in advance, and between two pointers the longer match wins	
[Char] — $\frac{1}{\ni}$ — P[Char] — E(extend°) — thin(prefix°×(T+T)) — P([Char]×Code) — P((X×𝟙)snoc) — P[Code] — est(R) — [Code] = Proposition 9.1: nil and extend have disjoint ranges. The decompositions of one string are $\frac{\mathtt{extend}^\circ}{\ni}$ $(\mathtt{ws}\#[a])=\{(\mathtt{ws},\mathtt{sym}\ a)\} \cup \{(\mathtt{xs},\mathtt{ptr}\ (\mathtt{ys},\mathtt{zs}))\mid \mathtt{xs}\#\mathtt{zs}=\mathtt{ws}\#[a],\ \mathtt{ys}\#\mathtt{zs}$ a proper prefix of $\mathtt{ws}\}$ — take the last character as a symbol, or end with a pointer	
[Char] — reduce — [[Char]×Code] — list((encode×𝟙)snoc) — [[Code]] — minlist(R) — [Code] ⊒ reduce implements $\frac{\mathtt{extend}^\circ}{\ni}$ $\mathtt{thin}(\mathtt{prefix}^\circ\times(T+T))$: thinning leaves at most two, the symbol and the pointer of the longest repeated tail lrt. Again exponential; the book gives no tabulation for it	

§9.4.1. Derivation

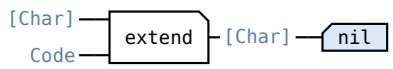
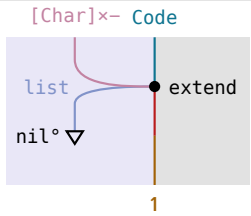
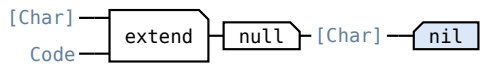
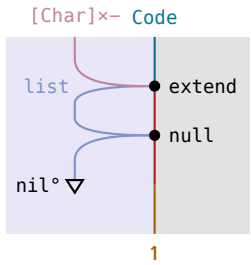
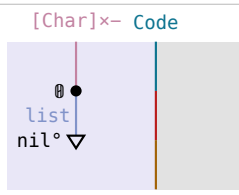
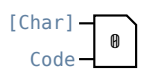
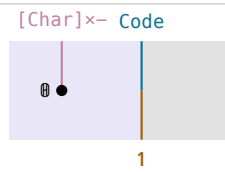
Q F(decode°) [nil,snoc] ⊑ F(decode°) [nil,snoc] R (9.4.1a)

every code sequence built from a decomposition **Q** puts above a given one — a code element of the same kind, a longer front string — costs at least as much as one built from the given one

circuit	Hinze–Marsden
	
=  `Q≜𝟙+(prefix°×(T+T))` is `F(T+T,prefix°)` — `l(val) Q l(val') ⟺ true, r(p) Q r(q) ⟺ (prefix°)(π₁(p),π₁(q)) ∧ (T+T)(π₂(p),π₂(q)), (∀val. ∀val'. u = l(val) ⟹ v = l(val') ⟹ false) ∧ (∀p. ∀q. u = r(p) ⟹ v = r(q) ⟹ false)` `⟹ (u Q v ⟺ false)` , `Q ≜ F((T+T),prefix°)`	
=  the bifunctor `F` preserves composition: `F(U,prefix°)F(𝟙,decode°)=F(U,prefix° decode°)`	
⊑  Proposition 9.4's second condition `prefix° decode°⊑decode° R`: dropping the last character drops the last `sym`, or shortens or drops the last pointer. Proved at `prefix` directly by induction, not through `init`	
=  `F` preserves composition: `F(U,decode° R)=F(𝟙,decode°)F(U,R)`	
⊑  Proposition 9.4's first condition `F(T+T,R)α⊑αR`, left as an exercise in the book: `snoc` adds the cost of the last element to both sides, and `[c,p](T+T)=[c,p]`	

`extend nil° = 0` (9.4.1b)

no string is both a result of **`extend`** and the result **`[]`** of **`nil`**

circuit	Hinze–Marsden
	
=  `nil` returns only `[]`, so `nil°=null nil°`	
= `extend null=0`: a symbol ends the string with a character, a pointer with its non-empty `zs`	
=  `0` composed with anything is `0`	

§10. Greedy Algorithms

§10.1. Theory

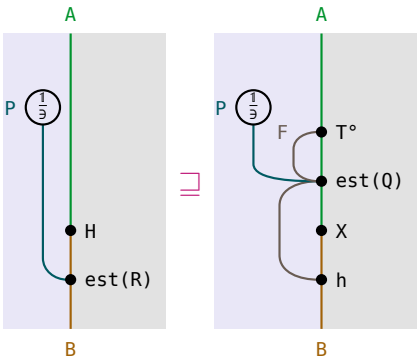
(10.1a)

h, T, R, H, M as in M ≜ $\frac{H}{\ni}$ est(R) ; additionally Q a **connected** preorder on the sets $\frac{T^\circ}{\ni}$ returns, so that $\frac{T^\circ}{\ni}$ est(Q) is entire.

Map(h) ∧ h Pres R ∧ R R ⊑ R ∧ ThinCondition(T,h,R,Q) ⟹ (μX : $\frac{T^\circ}{\ni}$ est(Q) F(X) h) ⊑ $\frac{H}{\ni}$ est(R) Theorem 10.1

the same optimum reached by keeping ONE decomposition at each step, so that no set is ever carried and the recursion runs on values alone

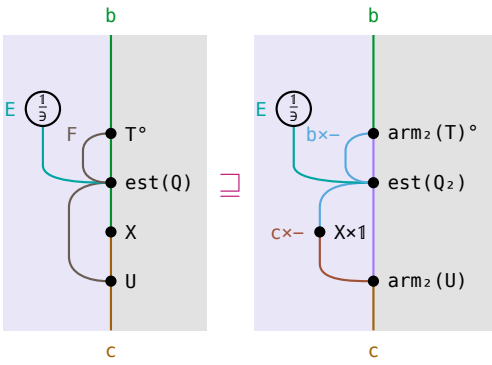
H≜⦇T⦈°⦇h⦈ — ≜ H, Q



(1) (2)

1	$\frac{H}{\ni}$ est(R)	
2	⊑ $\frac{T^\circ}{\ni}$ est(Q) F(X) h	

the body at T a coproduct of two arms

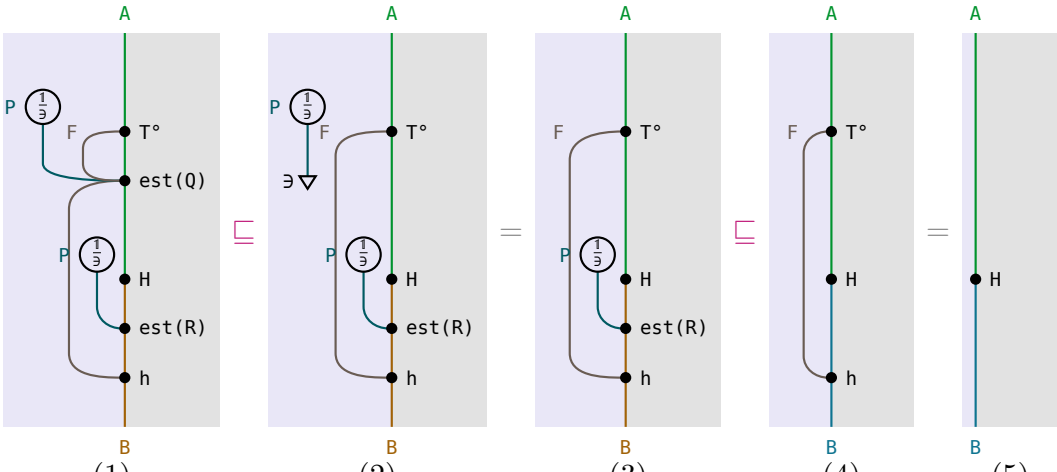


(1) (2)

1	$\frac{T^\circ}{\ni}$ est(Q) F(X) U	Proposition 10.1, V₂V₁°=𝟘
2	⊑ $\frac{arm_2(T)^\circ}{\ni}$ est(Q₂) (X × 𝟙) arm₂(U)	

$\frac{T^\circ}{\ni}$ est(Q) F($\frac{H}{\ni}$ est(R)) h ⊑ H (10.1b)

taking the input apart in one **Q**-extreme way, solving the parts by **M** and assembling by **h** returns only what **H** returns

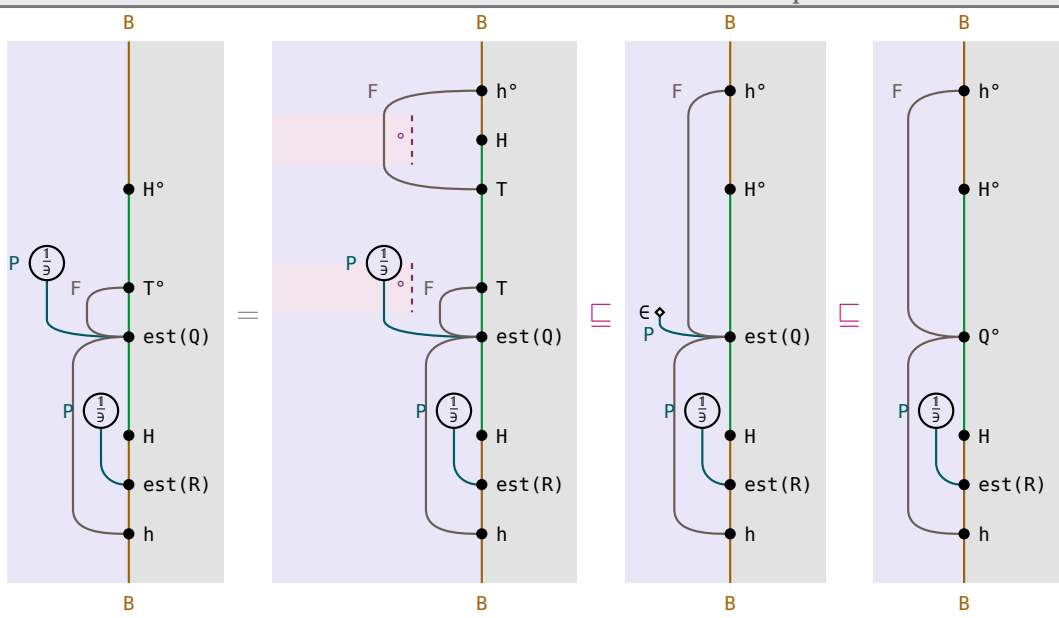


(1) (2) (3) (4) (5)

1	$\frac{T^\circ}{\ni}$ est(Q) F($\frac{H}{\ni}$ est(R)) h	est(Q) ⊑ ∋
2	⊑ $\frac{T^\circ}{\ni}$ ∋ F($\frac{H}{\ni}$ est(R)) h	$\frac{R}{\ni}$ ∋ = R
3	= T° F($\frac{H}{\ni}$ est(R)) h	$\frac{S}{\ni}$ est(R) ⊑ S
4	⊑ T° F(H) h	T° F(H) h = H
5	= H	

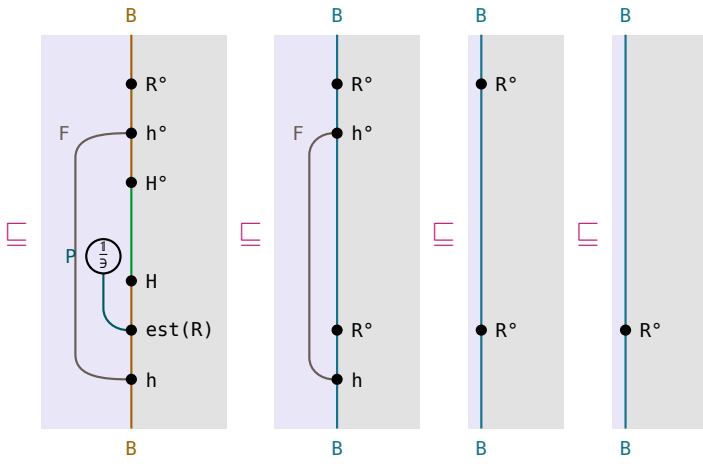
Map(h) ∧ h Pres R ∧ R R ⊑ R ∧ ThinCondition(T,h,R,Q) ⟹ H° $\frac{T°}{∋}$ est(Q) F($\frac{H}{∋}$ est(R)) h ⊑ R° (10.1c)

H° followed by the greedy body at M is ⊑R°: an answer of the body is never worse than an answer of H to the same input



(1) (2) (3) (4)

1	H° $\frac{T°}{∋}$ est(Q) F($\frac{H}{∋}$ est(R)) h	H° = h° F(H°) T
2	= h° F(H°) T $\frac{T°}{∋}$ est(Q) F($\frac{H}{∋}$ est(R)) h	T $\frac{T°}{∋}$ ⊑ ∈
3	⊑ h° F(H°) ∈ est(Q) F($\frac{H}{∋}$ est(R)) h	∈ est(R) ⊑ R°
4	⊑ h° F(H°) Q° F($\frac{H}{∋}$ est(R)) h	ThinCondition(T,h,R,Q) ⟹ h° F(H°) Q° ⊑ R° h° F(H°)

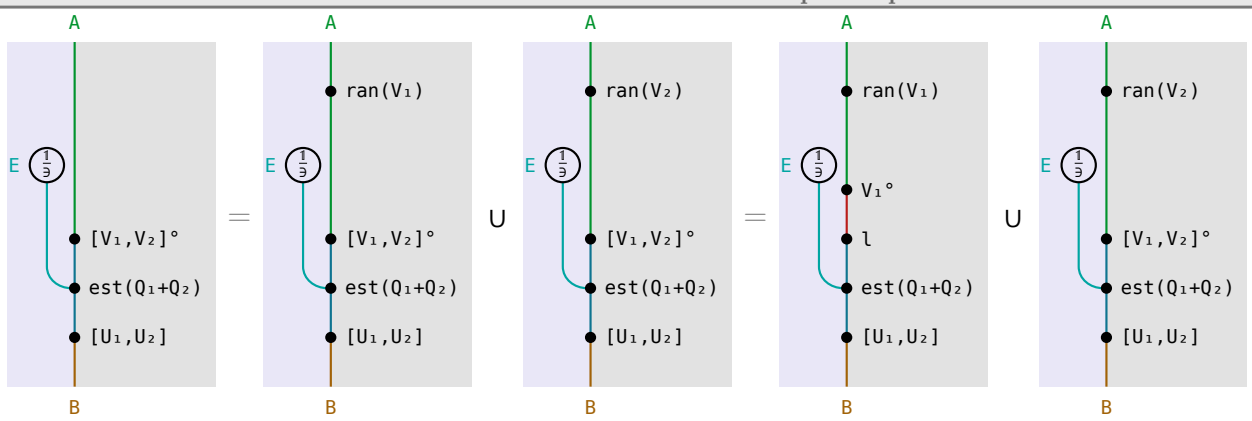


(5) (6) (7) (8)

5	⊑ R° h° F(H° $\frac{H}{∋}$ est(R)) h	S° $\frac{S}{∋}$ est(R) ⊑ R°
6	⊑ R° h° F(R°) h	Map(h) ∧ h Pres R ⟹ h° F(R°) h ⊑ R°
7	⊑ R° R°	R R ⊑ R ⟹ R° R° ⊑ R°
8	⊑ R°	

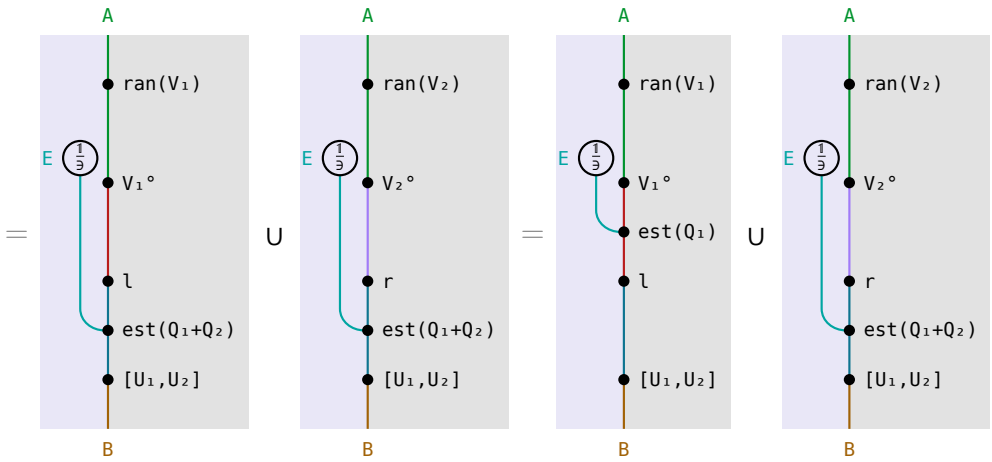
$V_2\ V_1{}^\circ = \emptyset \implies \frac{[V_1,V_2]^\circ}{\ni}\ est(Q_1 + Q_2)\ [U_1,U_2] = (ran(V_1)\ \frac{V_1{}^\circ}{\ni}\ est(Q_1)\ U_1) \cup (ran(V_2)\ \frac{V_2{}^\circ}{\ni}\ est(Q_2)\ U_2)$ Proposition 10.1

when V_1 and V_2 have disjoint ranges, the greedy step over $[V_1,V_2]$ runs the V_1 step on inputs V_1 reaches and the V_2 step on inputs V_2 reaches



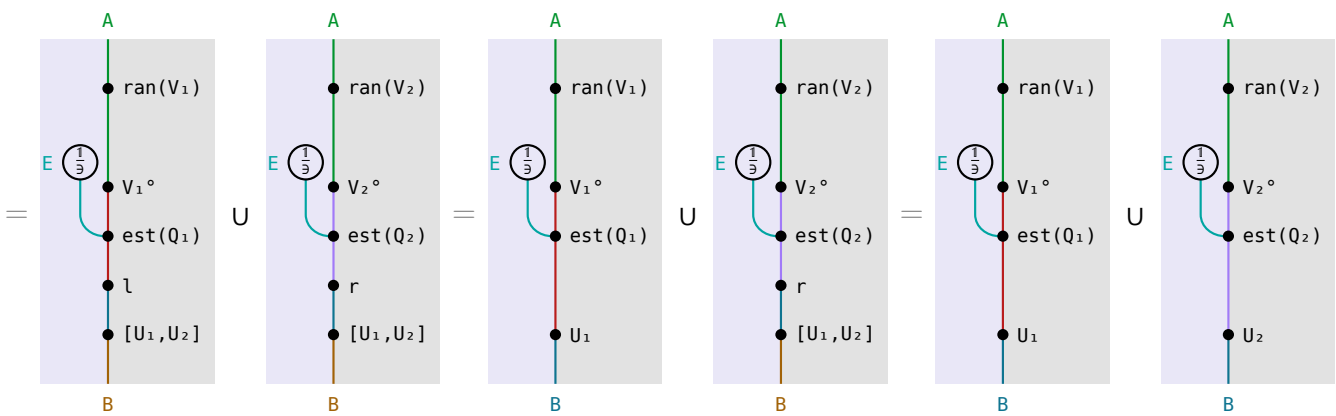
(1) (2) (3)

1	$\frac{[V_1,V_2]^\circ}{\ni}\ est(Q_1 + Q_2)\ [U_1,U_2]$	$\frac{[V_1,V_2]^\circ}{\ni}\ est(Q_1 + Q_2)\ [U_1,U_2] = (ran(V_1)\ \frac{[V_1,V_2]^\circ}{\ni}\ est(Q_1 + Q_2)\ [U_1,U_2]) \cup (ran(V_2)\ \frac{[V_1,V_2]^\circ}{\ni}\ est(Q_1 + Q_2)\ [U_1,U_2])$
2	$= (ran(V_1)\ \frac{[V_1,V_2]^\circ}{\ni}\ est(Q_1 + Q_2)\ [U_1,U_2]) \cup (ran(V_2)\ \frac{[V_1,V_2]^\circ}{\ni}\ est(Q_1 + Q_2)\ [U_1,U_2])$	$V_2\ V_1{}^\circ = \emptyset \implies ran(V_1)\ \frac{[V_1,V_2]^\circ}{\ni} = ran(V_1)\ \frac{V_1{}^\circ}{\ni}\ P(l)$
3	$= (ran(V_1)\ \frac{V_1{}^\circ}{\ni}\ P(l)\ est(Q_1 + Q_2)\ [U_1,U_2]) \cup (ran(V_2)\ \frac{[V_1,V_2]^\circ}{\ni}\ est(Q_1 + Q_2)\ [U_1,U_2])$	$V_2\ V_1{}^\circ = \emptyset \implies ran(V_2)\ \frac{[V_1,V_2]^\circ}{\ni} = ran(V_2)\ \frac{V_2{}^\circ}{\ni}\ P(r)$



(4) (5)

4	$= (ran(V_1)\ \frac{V_1{}^\circ}{\ni}\ P(l)\ est(Q_1 + Q_2)\ [U_1,U_2]) \cup (ran(V_2)\ \frac{V_2{}^\circ}{\ni}\ P(r)\ est(Q_1 + Q_2)\ [U_1,U_2])$	$P(l)\ est(Q_1 + Q_2) = est(Q_1)\ l$
5	$= (ran(V_1)\ \frac{V_1{}^\circ}{\ni}\ est(Q_1)\ l\ [U_1,U_2]) \cup (ran(V_2)\ \frac{V_2{}^\circ}{\ni}\ P(r)\ est(Q_1 + Q_2)\ [U_1,U_2])$	$P(r)\ est(Q_1 + Q_2) = est(Q_2)\ r$



(6) (7) (8)

6	$= (ran(V_1)\ \frac{V_1{}^\circ}{\ni}\ est(Q_1)\ l\ [U_1,U_2]) \cup (ran(V_2)\ \frac{V_2{}^\circ}{\ni}\ est(Q_2)\ r\ [U_1,U_2])$	$l\ [R,S] = R$
7	$= (ran(V_1)\ \frac{V_1{}^\circ}{\ni}\ est(Q_1)\ U_1) \cup (ran(V_2)\ \frac{V_2{}^\circ}{\ni}\ est(Q_2)\ r\ [U_1,U_2])$	$r\ [R,S] = S$
8	$= (ran(V_1)\ \frac{V_1{}^\circ}{\ni}\ est(Q_1)\ U_1) \cup (ran(V_2)\ \frac{V_2{}^\circ}{\ni}\ est(Q_2)\ U_2)$	

§10.2. The detab-entab problem

§10.2.1. Detab

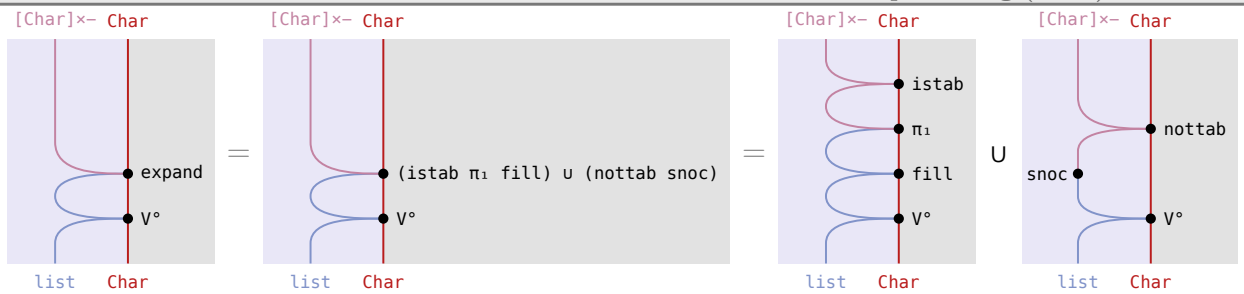
(10.2.1a)

	definition	type	meaning
1	⦇[nil,expand]⦈ = detab	String→String	replace every tab by blanks up to the next tab stop; H=detab°
2	expand(xs,a) ≜ if a = tb then fill(xs) else snoc(xs,a)	String	a tab fills the line to the next stop; any other character is appended
3	fill(xs) ≜ xs⧺blanks(n−col(xs) % n)	String	append blanks up to the next multiple of n columns
4	col(wrap(a)) = 0, col(snoc(x,a)) = if a = nl then 0 else col(x) + 1	String→ℕ	the column the string ends at: characters since the last newline
5	xs R ys ⟺ length(xs) ≤ length(ys)	[Char]→[Char]	x is no longer than y
6	a U b ⟺ a = tb ∨ a = b	Char→Char	a is the tab or a=b
7	xs V ys ⟺ prefix(xs,ys) ∧ fill(xs) = fill(ys)	[Char]→[Char]	a prefix that fills to the same string
8	Q ≜ 𝟙 + (V × U)	F[Char]→F[Char]	compare two decompositions by V on the strings and U on the characters
9	unfill(wrap(a)) = [], unfill(snoc(x,a)) = if a = blank ∧ col(snoc(x,a)) % n ≠ 0 then unfill(x) else snoc(x,a)	String→String	the shortest prefix that fills to the same string
10	tbc(wrap(a)) = 0, tbc(snoc(x,a)) = if a = blank ∧ col(snoc(x,a)) % n ≠ 0 then tbc(x) + 1 else 0	String→ℕ	the number of trailing blanks
11	triple ≜ ⟨unfill entab,⟨tbc,col⟩⟩	[Char]→String×(ℕ×ℕ)	what entab keeps: the unfilled output, its trailing blank count and its column

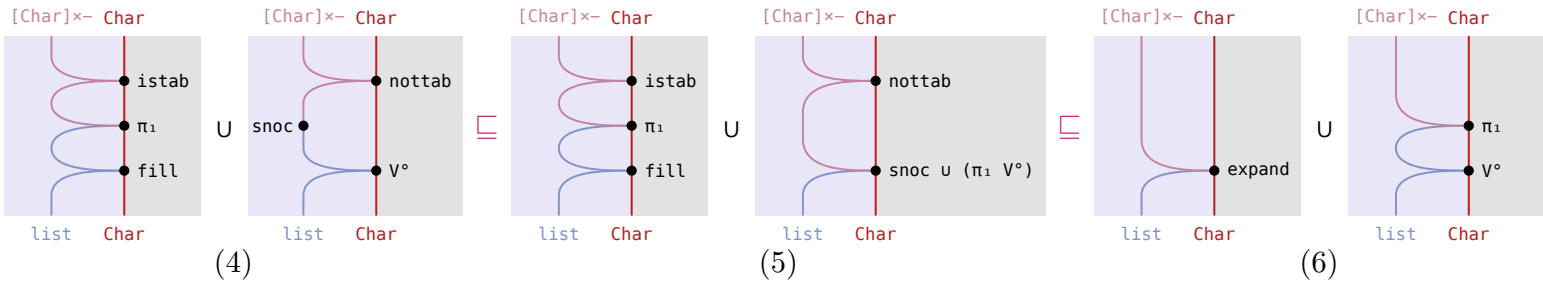
§10.2.2. Derivation

0 < n ⟹ expand V° ⊑ expand ∪ (π₁ V°) (10.2.2a)

shortening the output of one **expand** step to a **V**-smaller string either keeps the whole step (**expand**) or drops its character and shortens its input string (**π₁V°**)



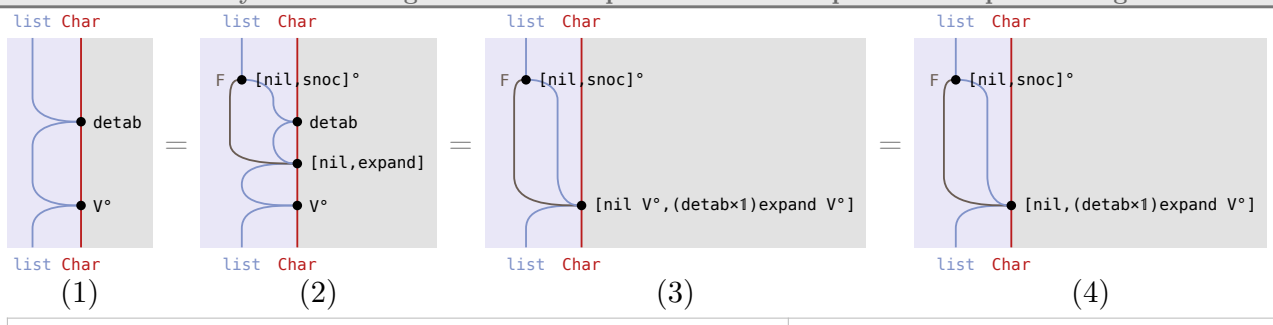
1	expand V°	expand = (istab π₁ fill) ∪ (nottab snoc)
2	= ((istab π₁ fill) ∪ (nottab snoc)) V°	(S ∪ T) R = (S R) ∪ (T R)
3	= (istab π₁ fill V°) ∪ (nottab snoc V°)	0 < n ∧ blank ≠ nl ⟹ fill V° = fill



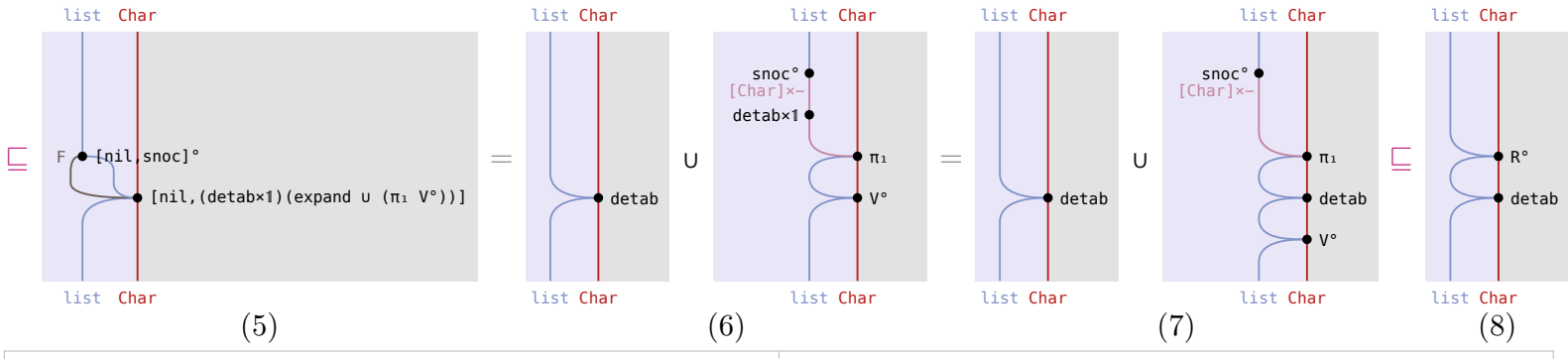
4	= (istab π₁ fill) ∪ (nottab snoc V°)	0 < n ∧ blank ≠ nl ⟹ snoc V° ⊑ snoc ∪ (π₁ V°)
5	⊑ (istab π₁ fill) ∪ (nottab (snoc ∪ (π₁ V°)))	(istab π₁ fill) ∪ (nottab (snoc ∪ X)) ⊑ expand ∪ X
6	⊑ expand ∪ (π₁ V°)	

0 < n ⟹ detab V° ⊑ R° detab (10.2.2b)

any **V**-shortening of **detab**'s output is **detab**'s output on an input no longer than the given one



1	detab V°	detab = [nil,snoc]° F(detab) [nil,expand]
2	= [nil,snoc]° F(detab) [nil,expand] V°	[R,S] Z = [R Z,S Z]
3	= [nil,snoc]° F(detab) [nil V°,expand V°]	nil V° = nil
4	= [nil,snoc]° F(detab) [nil,expand V°]	(10.2.2a) 0 < n ⟹ expand V° ⊑ expand ∪ (π₁ V°)

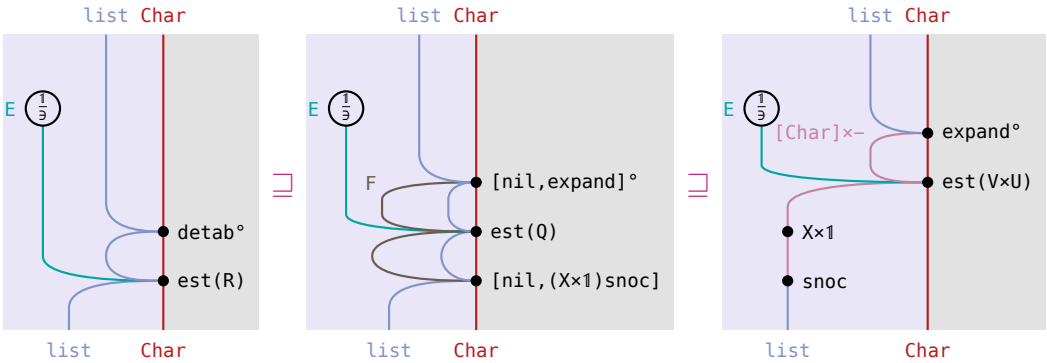


5	⊑ [nil,snoc]° F(detab) [nil,expand ∪ (π₁ V°)]	[nil,snoc]° F(detab) [nil,expand ∪ X] = detab ∪ (snoc° (detab × 𝟙) X)
6	= detab ∪ (snoc° (detab × 𝟙) π₁ V°)	snoc° (detab × 𝟙) π₁ Y = snoc° π₁ detab Y
7	= detab ∪ (snoc° π₁ detab V°)	0 < n ⟹ detab ∪ (snoc° π₁ detab V°) ⊑ R° detab
8	⊑ R° detab	

$$0 < n \implies (\mu X : \frac{[nil,expand]^\circ}{\ni}\ est(Q)\ F(X)\ [nil,snoc]) \sqsubseteq \frac{detab^\circ}{\ni}\ est(R) \qquad (10.2.2c)$$

the shortest input `detab` expands to the given output is one pass along that output, holding each blank back and cashing the held blanks in for a tab wherever the column reaches a tab stop

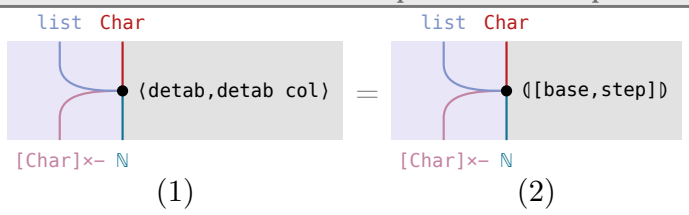
the specification — `detab(wrap(a)) = []`, `detab(snoc(x,a)) = expand(detab(x),a)` , `⦇[nil,expand]⦈ = detab` ; `detab entab=𝟙` and nothing shorter does



1	$\frac{detab^\circ}{\ni}$ `est(R)`	at `Q≜𝟙+(V×U)` : one character of input is decided at each step. `F(⊤,R)α⊑αR`. `detab prefix⊑R° detab` is FALSE , — at `n=8`, `detab [a,b,c,d,e,TB]=[a,b,c,d,e,BL,BL,BL]`, whose prefix `[a,b,c,d,e,BL,BL]` is longer than any input giving it, and `detab V°⊑R° detab` holds. `expand V°⊑expand ∪ (π₁V°)` — shortening the output either leaves the last step alone or discards it
2	$\sqsupseteq \frac{[nil,expand]^\circ}{\ni}$ `est(Q) F(X) [nil,snoc]`	Proposition 10.1: `nil` and `expand` have disjoint ranges. The greedy step is to emit a tab whenever a tab is legal, consuming all the blanks back to the previous tab stop
3	$\sqsupseteq \frac{expand^\circ}{\ni}$ `est(V × U) (X × 𝟙) snoc`	

$$\langle detab,detab\ col\rangle = ⦇[base,step]⦈ \qquad (10.2.2d)$$

one pass over the input carries the output so far together with its column



1	`⟨detab,detab col⟩`	both components are folds over the same input; `step` on `(detab x,col(detab x))` is `(detab,col detab)` one character further — not a tabulated row
2	`= ⦇[base,step]⦈`	

Exercise 10.1

- `⦇[base,step]⦈(convert(xs),r) ⟺ r = loop(step)(([],0),xs)`
  the fold of `[base,step]` over the input converted to a snoc-list is the left loop of `step` from `base`
- `π₁(loop(step)(([],c₀),xs)) = loop'((f,g),(c₀,xs))`
  Exercise 10.1: when `step` only appends `f(c,a)` to the output and moves the state by `g`, the output of the loop is `loop'(f,g)`, which never carries the output

(10.2.2e)

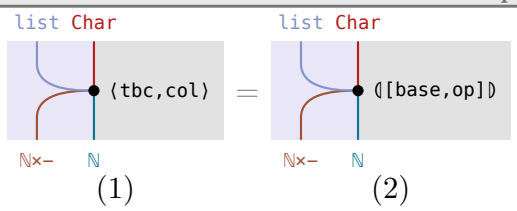
- `a ≠ tb ⟹ expand(ys,b) = snoc(xs,a) ⟺ fill(ys) = snoc(xs,a) ∧ b = tb ∨ (ys,b) = (xs,a)`
  `expand` produces `x#[a]` from a tab after a string that fills to it, or from `x` and `a`
- `0 < n ⟹ (∃ys. fill(ys) = snoc(xs,a)) ⟺ a = blank ∧ col(snoc(xs,a)) % n = 0`
  a string is a `fill` exactly when it ends in a blank on a tab stop

(10.1)

- `entab(xs) = entab(unfill(xs))#blanks(tbc(xs))`
  (10.1): the output of `entab` is that of `unfill x` followed by the trailing blanks held back
- `tbc[] = 0`
  the empty string has no trailing blanks
- `tbc(snoc(xs,a)) = if a = blank ∧ col(snoc(xs,a)) % n ≠ 0 then tbc(xs) + 1 else 0`
  a blank off a tab stop adds one held blank; anything else releases them all

⟨tbc,col⟩ = ⦇[base,op]⦈ (10.2.2f)

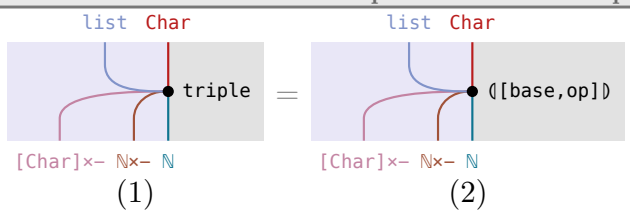
one pass counts the held blanks and the column together



1	⟨tbc,col⟩	op on (tbc x,col x) is (tbc,col) one character further, from the equations for tbc — (10.1) entab(xs) = entab(unfill(xs))#blanks(tbc(xs)) , tbc[] = 0 , tbc(snoc(xs,a)) = if a = blank ∧ col(snoc(xs,a)) % n ≠ 0 then tbc(xs) + 1 else 0
2	= ⦇[base,op]⦈	

triple = ⦇[base,op]⦈ (10.2.2g)

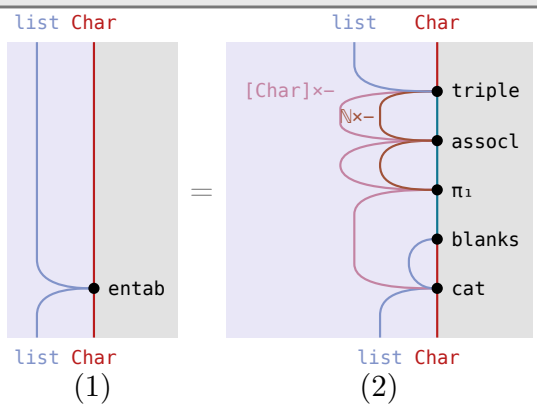
one pass carries the output up to the held blanks, their count and the column



1	triple	op on triple x is triple one character further: (10.1) where the blanks are released — (10.1) entab(xs) = entab(unfill(xs))#blanks(tbc(xs)) , tbc[] = 0 , tbc(snoc(xs,a)) = if a = blank ∧ col(snoc(xs,a)) % n ≠ 0 then tbc(xs) + 1 else 0
2	= ⦇[base,op]⦈	

entab = triple assocl π₁ (𝟙 × blanks) cat (10.2.2h)

entab is **triple** with the held blanks appended to its output



1	entab	(10.1) — (10.1) entab(xs) = entab(unfill(xs)) #blanks(tbc(xs)) , tbc[] = 0 , tbc(snoc(xs,a)) = if a = blank ∧ col(snoc(xs,a)) % n ≠ 0 then tbc(xs) + 1 else 0
2	= triple assocl π₁ (𝟙 × blanks) cat	

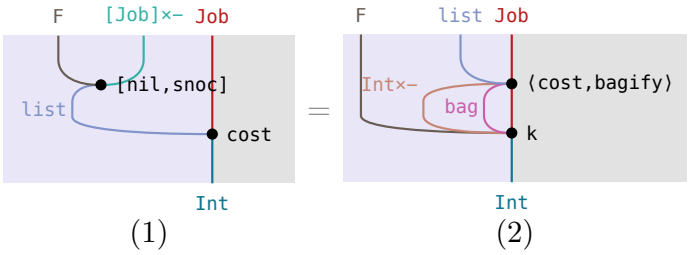
§10.3. The minimum tardiness problem

(10.3a)

	definition	type	meaning
1	bag(Job) ≜ Quotient(perm)	$\mathcal{A}$	a bag of jobs: a list up to reordering
2	snag(p) ≜ π₂(p) :: π₁(p)	bag(Job)	put one job into a bag
3	⦇[nil,snag]⦈ = bagify	[Job]→bag(Job)	forget the order of a schedule; H=bagify° lists every schedule of a bag
4	penalty(xs,j) ≜ (sum(list(ct))(xs) + ct(j)−dt(j))wt(j)	Int	how late job j finishes after the schedule xs, times its weight
5	cost(wrap(a)) = 0, cost(snoc(x,a)) = bmax(cost(x),penalty(x,a))	[Job]→Int	the largest penalty of any job in the schedule
6	bmax(a,b) ≜ if a ≤ b then b else a	Int	the larger of two numbers
7	R ≜ cost ≤ cost°	[Job]→[Job]	xs costs no more than ys
8	p add ys ⟺ add(π₂(p),ys)	[Job]×Job→[Job]	insert j anywhere in xs; perm≜bagify bagify°=⦇[nil,add]⦈
9	k ≜ [nil,assocr (𝟙×((bagify°×𝟙) penalty)) bmax]	F(Int×bag(Job))→ Int	the cost of a schedule from the cost of its front and the penalty of its last job
10	f(l(val)) = 0, f(r(p)) = (bagify°×𝟙) penalty(p)	F(bag(Job))→Int	the penalty of the last job, read off the bag before it
11	u Q v ⟺ f(u) ≤ f(v)	F(bag(Job))→ F(bag(Job))	the last job of u gets no larger penalty than that of v
12	p Q' q ⟺ (bagify°×𝟙) penalty(p) ≤ (bagify°×𝟙) penalty(q)	bag(Job)×Job→ bag(Job)×Job	Q on the snag summand: compare the penalties of the two last jobs
13	g ≜ [zero,penalty]	F[Job]→Int	(10.5): the penalty of the last job, zero for the empty schedule
14	m ≜ [zero,π₁ cost]	F[Job]→Int	(10.6), B&dM's h renamed as in (∀j. 0 ≤ ct(j)) ∧ (∀j. 0 ≤ wt(j)) ⟹ (μX : $\frac{\text{[nil,snag]°}}{\ni}$ est(Q) F(X) [nil,snoc]) ⊑ $\frac{\text{bagify°}}{\ni}$ est(R) : the cost of the schedule before the last job

[nil,snoc] cost = F(⟨cost,bagify⟩) k (10.3b)

the cost of a schedule is got from the cost and the bag of the schedule before its last job, and that job

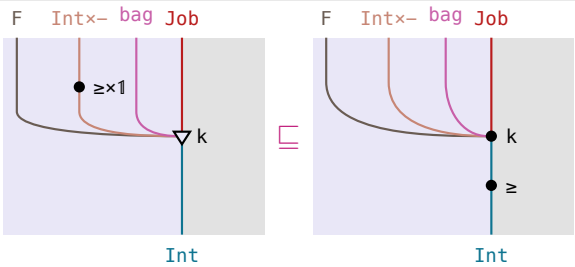


(1) (2)

1	[nil,snoc] cost	cost (xs#[j])=bmax (cost xs,penalty (xs,j)), and penalty(xs,j) reads only the bag of xs — penalty(xs,j) ≜ (sum(list(ct))(xs) + ct(j)−dt(j))wt(j)
2	= F(⟨cost,bagify⟩) k	

F(≥ × 𝟙) k ⊑ k ≥ Exercise 10.5

a larger cost before the last job gives a larger cost after it

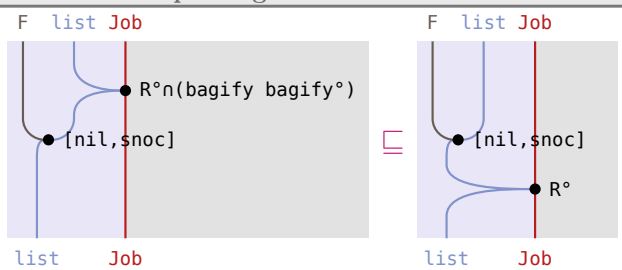


(1) (2)

1	F(≥ × 𝟙) k	bmax is monotone in its first argument
2	⊑ k ≥	

F(R° ∩ (bagify bagify°)) [nil,snoc] ⊑ [nil,snoc] R° (10.2)

improving the schedule before the last job, among schedules of the same bag, improves the whole schedule

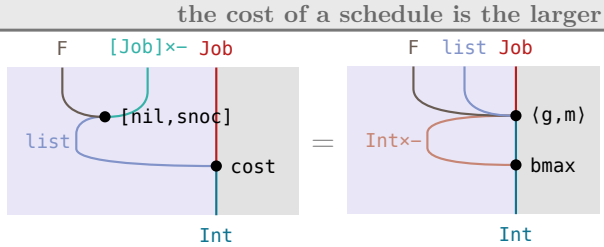


(1) (2)

1	F(R° ∩ (bagify bagify°)) [nil,snoc]	Proposition 9.3 at S≜bagify, ≤≜≥ — Proposition 9.3 Map(cost) ∧ Simple(S) ∧ h cost = F(⟨cost,S⟩) k ∧ F(≤ × 𝟙) k ⊑ k ≤ ⟹ F((cost ≤ cost°) ∩ (S S°)) h ⊑ h cost ≤ cost° , with [nil,snoc] cost = F(⟨cost,bagify⟩) k and Exercise 10.5 F(≥ × 𝟙) k ⊑ k ≥
2	⊑ [nil,snoc] R°	

[nil,snoc] cost = ⟨g,m⟩ bmax (10.4)

the cost of a schedule is the larger of the penalty of its last job and the cost of the schedule before it

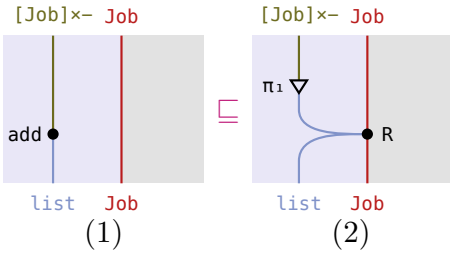


(1) (2)

1	[nil,snoc] cost	cost (xs#[j])=bmax (cost xs,penalty (xs,j)) and bmax commutes — cost(wrap(a)) = 0, cost(snoc(x,a)) = bmax(cost(x),penalty(x,a))
2	= ⟨g,m⟩ bmax	

(∀j. 0 ≤ ct(j)) ∧ (∀j. 0 ≤ wt(j)) ⟹ add ⊑ π₁ R (10.7)

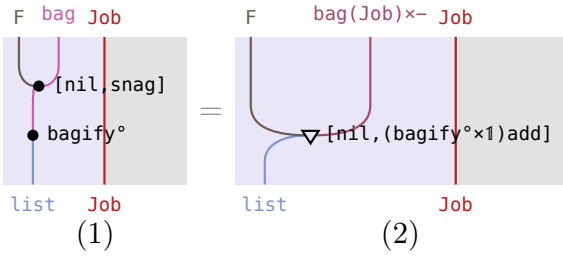
putting one more job anywhere into a schedule never lowers its cost



1	add	induction on where j goes: every job after it starts later, so no penalty falls
2	⊑ π₁ R	

[nil,snag] bagify° = [nil,(bagify° × 𝟙) add] (10.8)

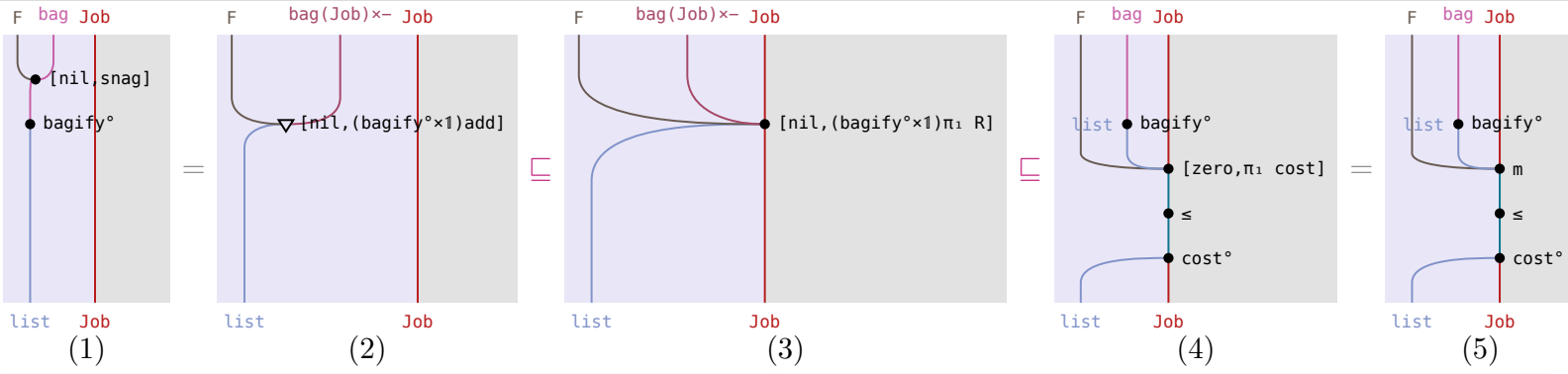
the orderings of a bag with one more job are the orderings of the bag with that job put in anywhere



1	[nil,snag] bagify°	take out the last copy of j and put it back — p add ys ⟺ add(π₂(p),ys)
2	= [nil,(bagify° × 𝟙) add]	

(∀j. 0 ≤ ct(j)) ∧ (∀j. 0 ≤ wt(j)) ⟹ [nil,snag] bagify° ⊑ F(bagify°) m ≤ cost° (10.3c)

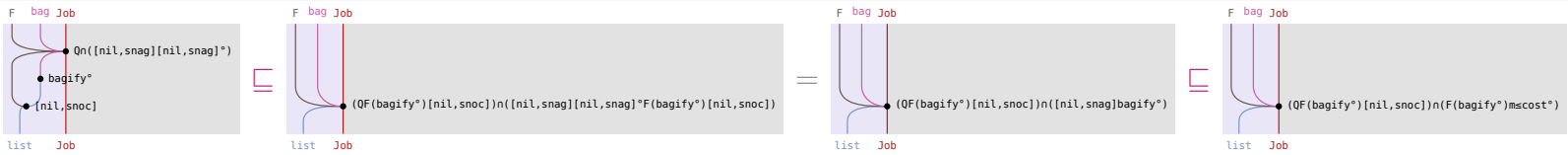
an ordering of a bag with one more job costs at least the cost of the ordering of the bag before its last job



1	[nil,snag] bagify°	(10.8) [nil,snag] bagify° = [nil,(bagify° × 𝟙) add]
2	= [nil,(bagify° × 𝟙) add]	(10.7) (∀j. 0 ≤ ct(j)) ∧ (∀j. 0 ≤ wt(j)) ⟹ add ⊑ π₁ R
3	⊑ [nil,(bagify° × 𝟙) π₁ R]	[nil,π₁ R] ⊑ [zero,π₁ cost] ≤ cost°
4	⊑ F(bagify°) [zero,π₁ cost] ≤ cost°	
5	= F(bagify°) m ≤ cost°	

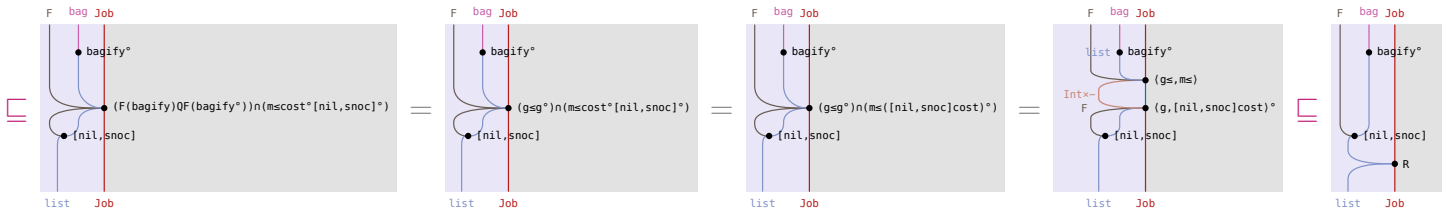
(∀j. 0 ≤ ct(j)) ∧ (∀j. 0 ≤ wt(j)) ⟹ (Q ∩ ([nil,snag] [nil,snag]°)) F(bagify°) [nil,snoc] ⊑ F(bagify°) [nil,snoc] R (10.3)

a schedule of a bag ending in a job of least penalty is no worse than any schedule of the same bag



(1) (2) (3) (4)

1	(Q ∩ ([nil,snag] [nil,snag]°)) F(bagify°) [nil,snoc]	(S ∩ T) R ⊑ (S R) ∩ (T R)
2	⊑ (Q F(bagify°) [nil,snoc]) ∩ ([nil,snag] [nil,snag]° F(bagify°) [nil,snoc])	[nil,snag]° F(bagify°) [nil,snoc] = bagify°
3	= (Q F(bagify°) [nil,snoc]) ∩ ([nil,snag] bagify°)	(10.3c) (∀j. 0 ≤ ct(j)) ∧ (∀j. 0 ≤ wt(j)) ⟹ [nil,snag] bagify° ⊑ F(bagify°) m ≤ cost°
4	⊑ (Q F(bagify°) [nil,snoc]) ∩ (F(bagify°) m ≤ cost°)	(Q F(bagify°) [nil,snoc]) ∩ (F(bagify°) M) ⊑ F(bagify°) ((F(bagify) Q F(bagify°)) ∩ (M [nil,snoc]°)) [nil,snoc]

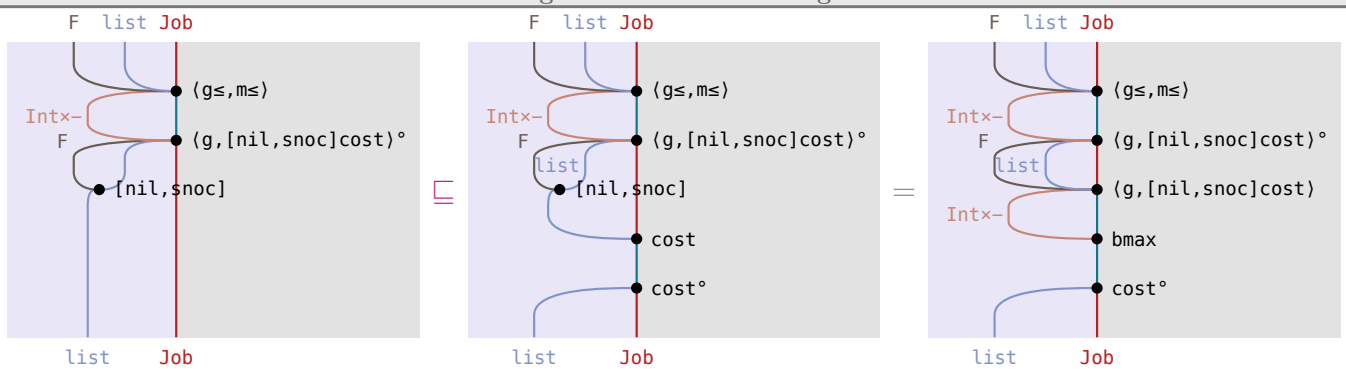


(5) (6) (7) (8) (9)

5	⊑ F(bagify°) ((F(bagify) Q F(bagify°)) ∩ (m ≤ cost° [nil,snoc]°)) [nil,snoc]	F(bagify) Q F(bagify°) = g ≤ g°
6	= F(bagify°) ((g ≤ g°) ∩ (m ≤ cost° [nil,snoc]°)) [nil,snoc]	(R S)° = S° R°
7	= F(bagify°) ((g ≤ g°) ∩ (m ≤ ([nil,snoc] cost)°)) [nil,snoc]	⟨X,Y⟩ ⟨R,S⟩° = (X R°) ∩ (Y S°)
8	= F(bagify°) ⟨g ≤,m ≤⟩ ⟨g,[nil,snoc] cost⟩° [nil,snoc]	(10.3d) ⟨g ≤,m ≤⟩ ⟨g,[nil,snoc] cost⟩° [nil,snoc] ⊑ [nil,snoc] R
9	⊑ F(bagify°) [nil,snoc] R	

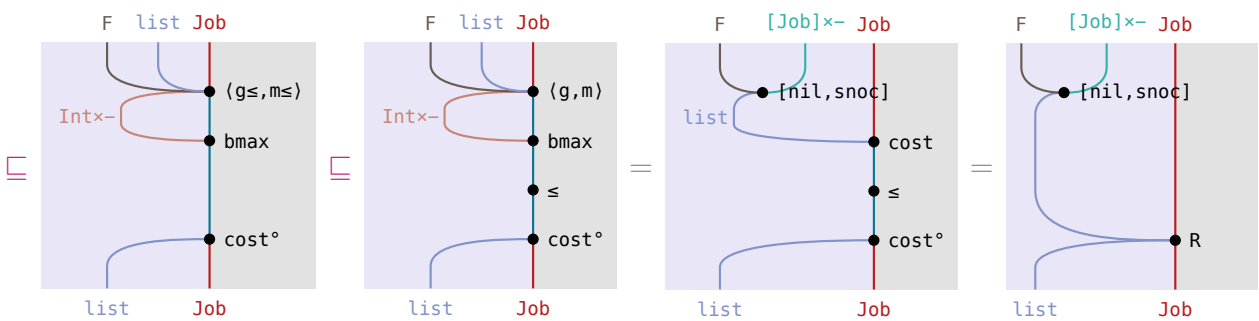
⟨g ≤,m ≤⟩ ⟨g,[nil,snoc] cost⟩° [nil,snoc] ⊑ [nil,snoc] R (10.3d)

a job whose penalty is at most the last penalty, put after a schedule costing at most the cost before the last job, gives a schedule costing at most the whole



(1) (2) (3)

1	⟨g ≤,m ≤⟩ ⟨g,[nil,snoc] cost⟩° [nil,snoc]	𝟙 ⊑ cost cost°
2	⊑ ⟨g ≤,m ≤⟩ ⟨g,[nil,snoc] cost⟩° [nil,snoc] cost cost°	[nil,snoc] cost = ⟨g,[nil,snoc] cost⟩ bmax
3	= ⟨g ≤,m ≤⟩ ⟨g,[nil,snoc] cost⟩° ⟨g,[nil,snoc] cost⟩ bmax cost°	⟨g,[nil,snoc] cost⟩° ⟨g,[nil,snoc] cost⟩ ⊑ 𝟙



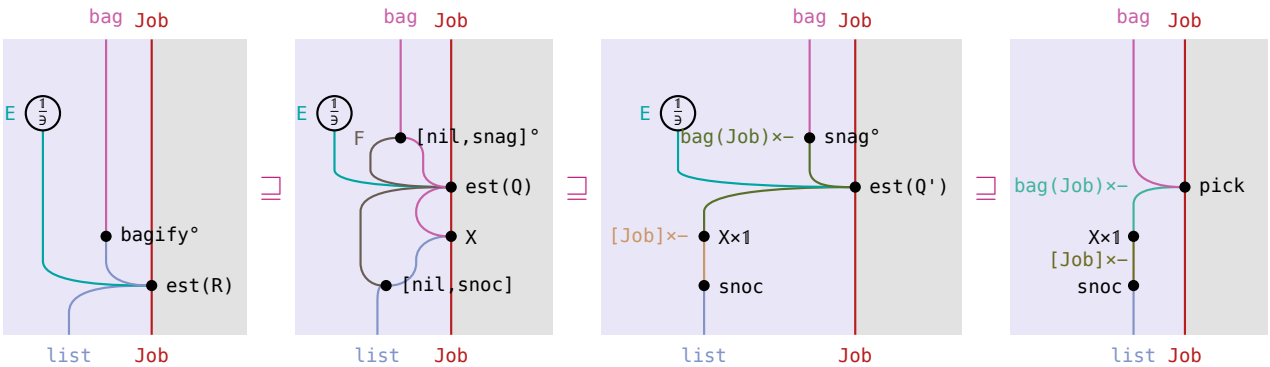
(4) (5) (6) (7)

4	⊑ ⟨g ≤,m ≤⟩ bmax cost°	⟨g ≤,m ≤⟩ bmax ⊑ ⟨g,m⟩ bmax ≤
5	⊑ ⟨g,m⟩ bmax ≤ cost°	(10.4) [nil,snoc] cost = ⟨g,m⟩ bmax
6	= [nil,snoc] cost ≤ cost°	R = cost ≤ cost°
7	= [nil,snoc] R	

$$\texttt{schedule=(null→nil,pick (schedule×𝟙) snoc),}\ \ (\forall j.\ 0 \le \texttt{ct(j)}) \wedge (\forall j.\ 0 \le \texttt{wt(j)}) \Longrightarrow (\mu X : \frac{\texttt{[nil,snag]°}}{\ni}\ \texttt{est(Q) F(X) [nil,snoc])} \sqsubseteq \frac{\texttt{bagify°}}{\ni}\ \texttt{est(R)} \tag{10.3e}$$

an ordering of the given bag with least maximum penalty is got by taking a job of least penalty out of the bag, putting it last, and scheduling what is left the same way

the specification — `⦇[nil,snag]⦈ = bagify`



1	$\frac{\texttt{bagify°}}{\ni}$ `est(R)`	No greedy **reduce** exists — one would also solve every prefix of the input, and the best schedule of a prefix need not extend to a best schedule of the whole
2	$\sqsupseteq \frac{\texttt{[nil,snag]°}}{\ni}$ `est(Q) F(X) [nil,snoc]`	Proposition 10.1: `nil` and `snag` have disjoint ranges
3	$\sqsupseteq \frac{\texttt{snag°}}{\ni}$ `est(Q') (X × 𝟙) snoc`	`pick`$\sqsubseteq \frac{\texttt{snag°}}{\ni}$ `est(Q')`, a partial function, quadratic in the number of jobs
4	$\sqsupseteq$ `pick (X × 𝟙) snoc`	

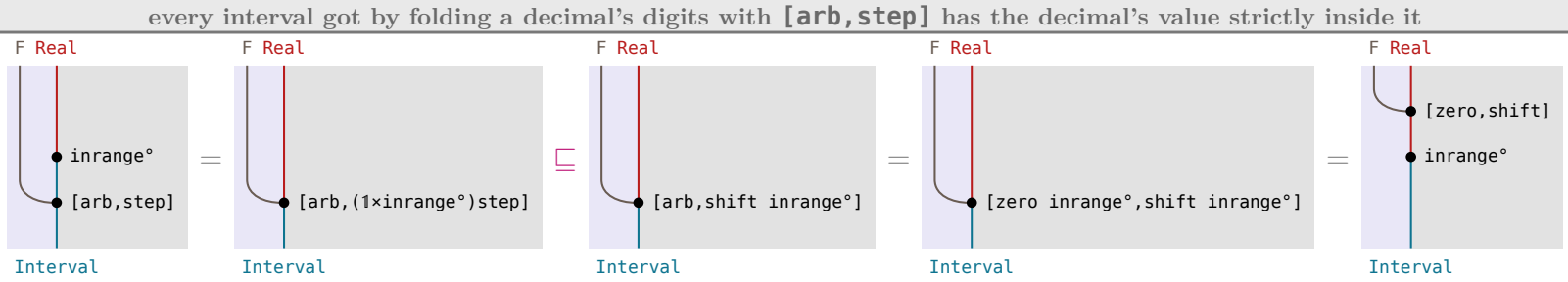
§10.4. The TeX problem

(10.4a)

	definition	type	meaning
1	`intern ≜ val round`	`[Digit]→[0,2¹⁶)`	the value of a decimal fraction, rounded to `16` binary places
2	`val ≜ ⦇[zero,shift]⦈`	`[Digit]→Real`	the real number a digit list denotes after the decimal point
3	`shift(d,r) ≜ (d + r)/10`	`Digit×Real→Real`	put digit `d` in front of `r`: `(d+r)/10`
4	`round° = interval inrange`	`Real→[0,2¹⁶)`	the nearest integer to `2¹⁶r`: `round r=n` iff `2n−1<2¹⁷r<2n+1`
5	`interval(n) ≜ ((2n−1)/w,(2n + 1)/w)`	`[0,2¹⁶)→ Interval`	the open interval `((2n−1)/2¹⁷, (2n+1)/2¹⁷)` of the reals that round to `n`
6	`p inrange r ⟺ lo(p) < r ∧ r < hi(p)`	`Interval→Real`	`r` lies strictly inside `(a,b)`
7	`Legal(p) ≜ 0 < hi(p) ∧ hi(p) < 1 ∧ lo(p) < hi(p)`	`Prop`	(10.9): `0<b<1` and `a<b`; `Interval` is the legal pairs
8	`x arb p ⟺ lo(p) < 0 ∧ 0 < hi(p)`	`1→Interval`	any legal interval
9	`step(p) ≜ (shift(π₁(p),lo(π₂(p))),shift(π₁(p),hi(π₂(p))))`	`Digit×Interval→ Interval`	put digit `d` in front of both ends: `((d+a)/10, (d+b)/10)`
10	`H ≜ ⦇[arb,step]⦈°`	`Interval→ [Digit]`	every digit list whose interval the given one is
11	`xs R ys ⟺ length(xs) ≤ length(ys)`	`[Digit]→[Digit]`	`x` is no longer than `y`
12	`l ≜ inl`	`1→FX`	the injection of the empty case into `FX=1+(Digit×X)`
13	`r ≜ inr`	`Digit×X→FX`	the injection of the digit case into `FX=1+(Digit×X)`
14	`! ≜ ⊸ ()`	`Digit×Interval→ 1`	the one map to `1`
15	`Q ≜ (l° !° r) ∪ 𝟙`	`F(Interval)→ F(Interval)`	`u` is the empty case and `z` a digit case, or `u=z`
16	`w ≜ 131072`	`Int`	`2¹⁷`

⦇[arb,step]⦈ ⊑ val inrange° (10.4b)

every interval got by folding a decimal's digits with **[arb,step]** has the decimal's value strictly inside it

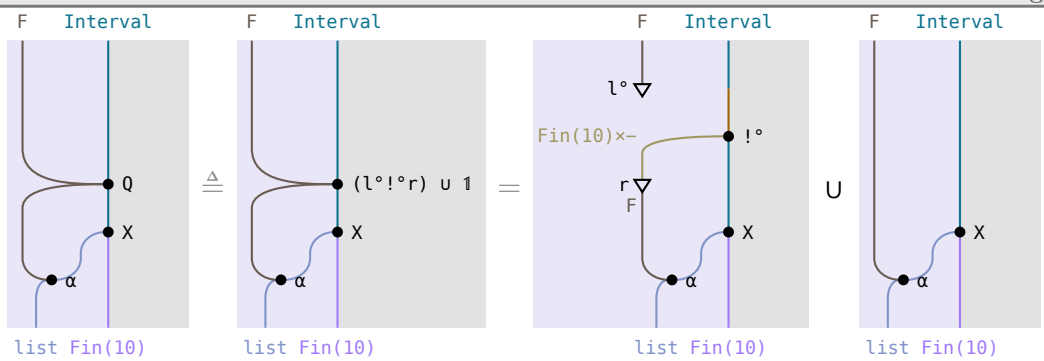


(1) (2) (3) (4) (5)

1	`F(inrange°) [arb,step]`	`F(S) [T,U] = [T,(𝟙 × S) U]`
2	`= [arb,(𝟙 × inrange°) step]`	`(𝟙 × inrange°) step ⊑ shift inrange°`
3	`⊑ [arb,shift inrange°]`	`zero inrange° = arb`
4	`= [zero inrange°,shift inrange°]`	`[R,S] Z = [R Z,S Z]`
5	`= [zero,shift] inrange°`	

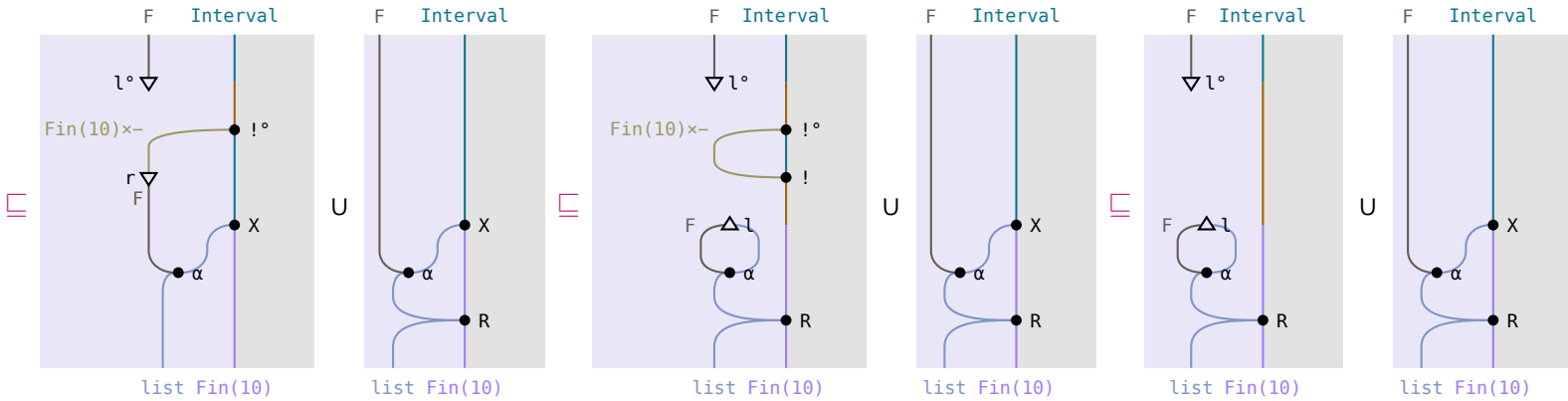
Q F(X) α ⊑ F(X) α R (10.4c)

choosing by Q among the one-step decompositions before building with F(X) and α yields only decimals that are R-related to one built without choosing



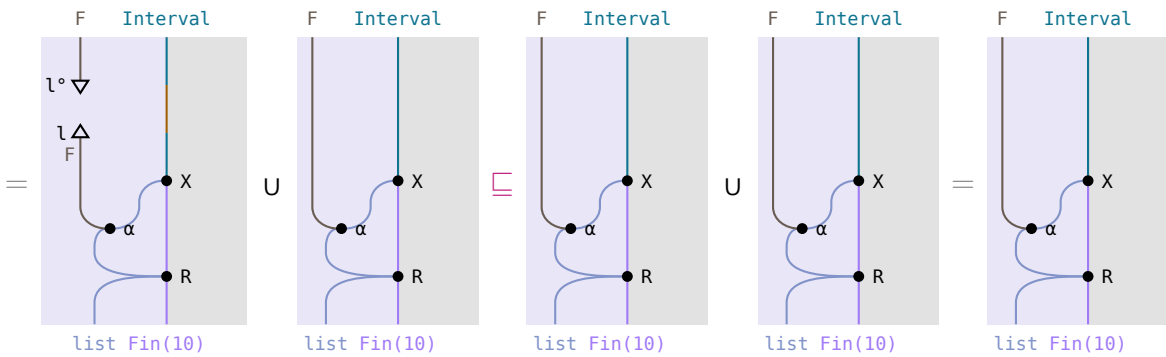
(1) (2) (3)

1	Q F(X) α	Q ≜ (l° !° r) ∪ 𝟙
2	≜ ((l° !° r) ∪ 𝟙) F(X) α	(S ∪ T) R = (S R) ∪ (T R)
3	= (l° !° r F(X) α) ∪ (F(X) α)	𝟙 ⊑ R



(4) (5) (6)

4	⊑ (l° !° r F(X) α) ∪ (F(X) α R)	r F(X) α ⊑ ! l α R
5	⊑ (l° !° ! l α R) ∪ (F(X) α R)	!° ! ⊑ 𝟙
6	⊑ (l° l α R) ∪ (F(X) α R)	l F(X) = l

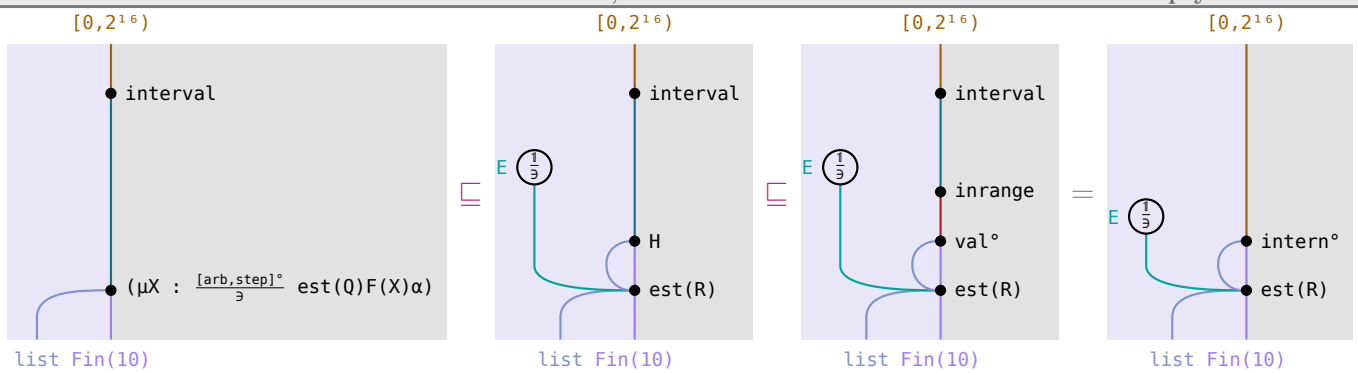


(7) (8) (9)

7	= (l° l F(X) α R) ∪ (F(X) α R)	l° l ⊑ 𝟙
8	⊑ (F(X) α R) ∪ (F(X) α R)	R ∪ R = R
9	= F(X) α R	

extern(n)=f(2n−1,2n+1), interval (μX : $\frac{\text{[arb,step]°}}{\ni}$ est(Q) F(X) α) ⊑ $\frac{\text{intern°}}{\ni}$ est(R) (10.4d)

a shortest decimal whose internal representation is the given multiple of 2^{-16} is got by emitting the one digit the interval of admissible reals allows, until that interval contains zero and the empty decimal will do



(1) (2) (3) (4)

1	interval (μX : $\frac{\text{[arb,step]°}}{\ni}$ est(Q) F(X) α)	$\frac{\text{[arb,step]°}}{\ni}$ est(Q) F($\frac{\text{H}}{\ni}$ est(R)) α ⊑ $\frac{\text{H}}{\ni}$ est(R)
2	⊑ interval $\frac{\text{H}}{\ni}$ est(R)	$\frac{\text{H}}{\ni}$ est(R) ⊑ $\frac{\text{inrange val°}}{\ni}$ est(R)
3	⊑ interval $\frac{\text{inrange val°}}{\ni}$ est(R)	$\frac{\text{intern°}}{\ni}$ = interval $\frac{\text{inrange val°}}{\ni}$
4	= $\frac{\text{intern°}}{\ni}$ est(R)	

(10.4e)

- `f(a,b) = xs ⟺ a < 0 ∧ xs = [] ∨ ¬a < 0 ∧ (∃d. ∃ys. d = ⌊10b⌋ ∧ f(10a−d,10b−d) = ys ∧ xs = [d]⧺ys)`
  `f` gives `(a,b)` the empty decimal when `a<0`, and otherwise the digit `d=⌊10b⌋` followed by what it gives `(10a−d,10b−d)`
- `(μX : [arb,step]°/∋ est(Q) F(X) α) = f`
  the least solution of the greedy recursion of `interval (μX : [arb,step]°/∋ est(Q) F(X) α) ⊑ intern°/∋ est(R)` is `f`
- `d = ⌊10b⌋ ∧ e = ⌊10b⌋ ⟹ d = e`
  Exercise 10.12: two digits `d` and `e` with `0<10b−d<1` and `0<10b−e<1` are equal
- `(0 < q−p ∧ q < w) ∧ p < 0 ⟹ f(p,q) = []`
  the program's first clause: `f(p,q)=[]` when `p<0`
- `(0 < q−p ∧ q < w) ∧ ¬p < 0 ⟹ f(p,q) = [10q/w]⧺f(10p−w·(10q/w),10q−w·(10q/w))`
  the program's second clause: with `d=(10q) div w`, `f(p,q)` is `d` followed by `f(10p−w·d,10q−w·d)`
- `(0 < q−p ∧ q < w) ∧ Reach(p,q) ⟹ f(p/w,q/w) = f(p,q)`
  on the pairs the program reaches from `interval n`, the integer `f` at `(p,q)` is a value of the rational `f` at `(p/w,q/w)`
- `interval f = extern`
  `f` after `interval` is the program `extern`, whose value at `n` is `f(2n−1,2n+1)`

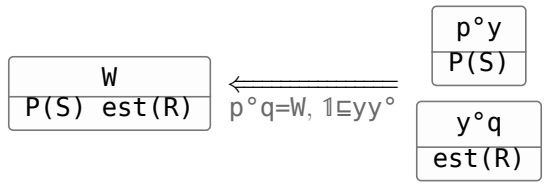
§A. Appendix

§A.1. P(S) est(R)=(∋S)∩(∈\(SR°))

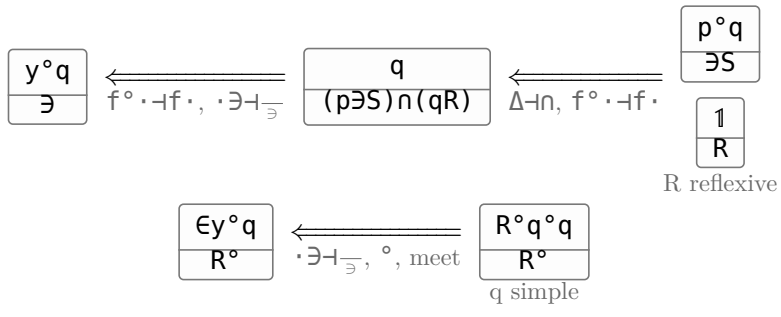
(A.1a)

(p,q) tabulates W≜(∋S)∩(∈\(SR°)), and y≜ $\frac{(p\ni S)\cap(qR)}{\ni}$, a map.

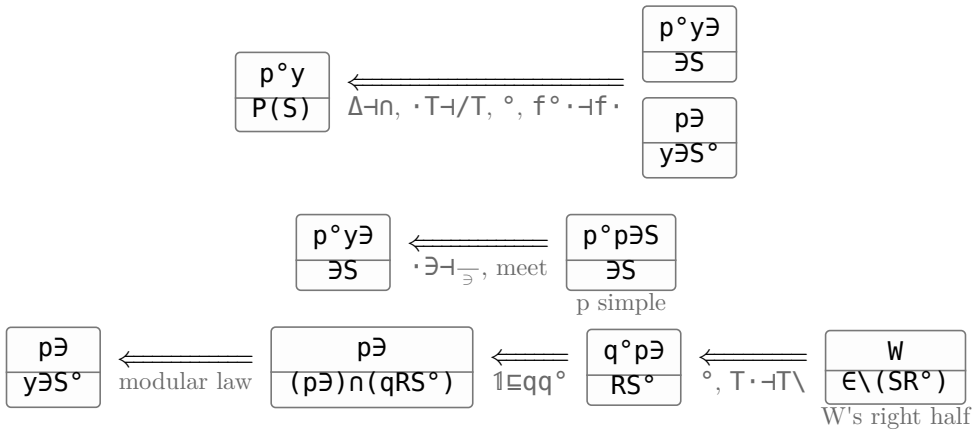
(A.1b)



(A.1c)



(A.1d)



§A.2. P(est(R)) est(R)=P(dom(est(R))) union est(R)

(A.2a)

P(est(R)) est(R) ══ dom(est(R)) est(R)=est(R) ══ P(dom(est(R)) est(R)) est(R)

══ P a relator ══ P(dom(est(R)))P(est(R)) est(R)

(7.11) R ⊑ R ⊑ R
⟹ P(est(R)) est(R) ⊑ union est(R)

P(dom(est(R))) union est(R)

(A.2b)

P(dom(est(R))) union est(R)
∋est(R)

⇐ 𝟙 ⊑ R ⟹ P(S) est(R) = (∋ S) ∩ (∈ \ (S R°)) at S:=est(R), Δ⊣∩, T·⊣T\

P(dom(est(R))) union est(R)
P(est(R)) est(R)

∈P(dom(est(R))) union est(R)
est(R)R°

P(dom(est(R))) union est(R)
∋est(R)

⇐ P(dom(est(R)))⊑𝟙

union est(R)
∋est(R)

⇐ T(U∩V)⊑TU∩TV, ·∋⊣∋, (7.3) union (∈ \ R) = ∈ \ (∈ \ R)

(∋∋)∩(∈\(∈\R°))
∋est(R)

⇐ modular law

∋(∋∩(∈(∈\(∈\R°))))
∋(∋∩(∈\R°))

counit of T · ⊣T\

∈P(dom(est(R))) union est(R)
est(R)R°

⇐ ∈ lax natural

dom(est(R))∈union est(R)
est(R)R°

⇐ dom(est(R))⊑est(R) est(R)°

est(R) est(R)°∈union est(R)
est(R)R°

⇐ est(R)°⊑∈, ∈∈union⊑∈

est(R)∈est(R)
est(R)R°

UP of est

§A.3. Shortest paths on a cylinder, on lists

(A.3a)

	definition	type	note
1	`F(A,X)=A+A×X`	$\mathcal{A}\to\mathcal{A}$	The base functor: one square of the new column, alone or in front of the path so far.
2	`L=list⁺`	$\mathcal{A}\to\mathcal{A}$	A path is a non-empty list of squares, one per column crossed.
3	`α` the initial algebra	`F(μF)→μF`	`α(5)=[5]` and `α(1,[5])=[1,5]`: a path is started by one square, or extended by one.
4	`N` the `n`-tuple relator	$\mathcal{A}\to\mathcal{A}$	One component per row of a column, so the fold carries `n` answers at once.
5	`R≜sum≤sum°`	`[ℕ]→[ℕ]`	The cost of a path, which the cheapest minimises.
6	`setify`	`A[n]→PA`	`setify(1,2,3,4)={1,2,3,4}` — which row a component came from is forgotten.
7	`moves`	`A[n]→P(A[n])`	`moves(x)={up(x),x,down(x)}` — rotated up, unrotated, rotated down.
8	`trans`	`P(A[n])→(P A)[n]`	`trans{(a,b,c),(x,y,z)}=({a,x},{b,y},{c,z})` — component `k` of the result is the set of the `k`-th components.
9	`zip`	`F(a[n])→(F a)[n]`	`zip((1,2,3,4),({[5]},{[6]},{[7]},{[8]}))=((1,{[5]}),(2,{[6]}),(3,{[7]}),(4,{[8]})).`
10	`cp≜` $\frac{\texttt{F(1,∋)}}{\texttt{∋}}$	`F(PA)→PFA`	`cp(1,{[5],[6],[8]})={(1,[5]),(1,[6]),(1,[8])}.`
11	`gen ≜ F(𝟙,moves trans N(union))` `zip N(cp(F(A,−)) E(α))`	`F(NA,N(P(TA)))→` `N(P(TA))`	`gen((1,2,3,4),({[5]},{[6]},{[7]},{[8]}))` is worked out in `gen ≜ F(𝟙,moves trans N(union))` `zip N(cp(F(A,−)) E(α))` .
12	`paths ≜ ⦇gen⦈ setify union`	`T(NA)→P(TA)`	`paths[(1,2,3,4),(5,6,7,8)]` is the union of `gen ≜ F(𝟙,moves trans N(union)) zip N(cp(F(A,−)) E(α))` ‘s four sets: 12 paths, 3 from each entry row.
13	the specification `paths est(R)`	`[ℕ[n]]→[ℕ]`	A cheapest path from the entry side to the exit side.

§A.3.1. N(E(L A))

(A.3.1a)

```
1  5
2  6      [(1,2,3,4),(5,6,7,8)] : L N Nat
3  7
4  8
```

(A.3.1b)

```
gen ≜ F(𝟙,moves trans N(union)) zip N(cp(F(A,−)) E(α))

⦇gen⦈[(5,6,7,8)] = ({[5]},{[6]},{[7]},{[8]})

gen((1,2,3,4),({[5]},{[6]},{[7]},{[8]})) =
    ( {[1,5],[1,6],[1,8]},
      {[2,5],[2,6],[2,7]},
      {[3,6],[3,7],[3,8]},
      {[4,5],[4,7],[4,8]} )
```

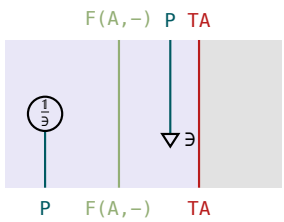
(A.3.1c)

1	N	the 4-tuple, one component per row of the entry column
2	E	the set of paths that can start in that row
3	L	a path is the non-empty list of the squares it crosses

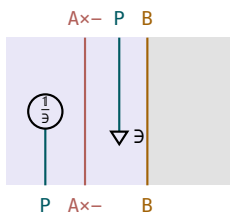
row 1's set has no `[1,7]` — from row 1 only rows 4, 1 and 2 are reachable — and it does have `[1,8]`, because the cylinder glues the bottom row to the top. That is `moves trans N(union)` of `gen ≜ F(𝟙,moves trans N(union)) zip N(cp(F(A,−)) E(α))` : component `k` collects the paths of rows `k-1`, `k`, `k+1`.

§A.3.2. The cross product cp≜$\frac{F(\mathbb{1},\ni)}{\ni}$

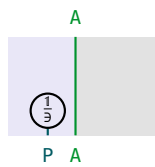
(A.3.2a)



cp on all of F(A,E(L A))



the A×− summand: ∋ picks one path, 𝟙%∋ collects the results



the A summand, no E to distribute: 𝟙%∋ alone, a↦{a}

(A.3.2b)

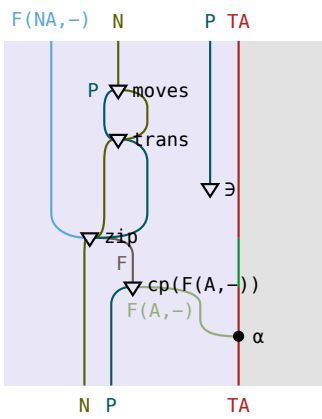
	type	note
1	F(A,−)=A+A×−	one wire carries the whole functor, both summands with it; the ∋ under it is F(𝟙,∋)
2	∋:E(L A)→L A	the ∋ inside F(𝟙,∋): a set of paths, one of them — L A is one path, A one square
3	F(𝟙,∋)=𝟙+𝟙×∋	:A+A×E(L A)→A+A×L A, the relator acting on each summand
4	∋:E(A+A×L A)→A+A×L A	the ∋ under the bar: a set of cells, one of them — a different ∋
5	cp:A+A×E(L A)→E(A+A×L A)	$\frac{R}{\ni}$ turns R:X→Y into X→E Y
6	cp=$\frac{\mathbb{1}}{\ni}$ E(F(𝟙,∋))	what the picture draws: the unit makes the outer E, leaving one ∋ — $\frac{R}{\ni}$ = $\frac{\mathbb{1}}{\ni}$ E(R)

(A.3.2c)

```
w      = (1,{[5],[6],[8]})            : A×E(L A)     the right summand
𝟙×∋    : A×E(L A) → A×L A                            𝟙 keeps the square 1, ∋ picks one path
         w ↦ (1,[5]), (1,[6]), (1,[8]) : A×L A        one output per path in w
cp(w)  = {(1,[5]),(1,[6]),(1,[8])}    : E(A+A×L A)   the three of them, collected
```

§A.3.3. gen=F(𝟙,moves trans N(union)) zip N(cp P(α))

(A.3.3a)



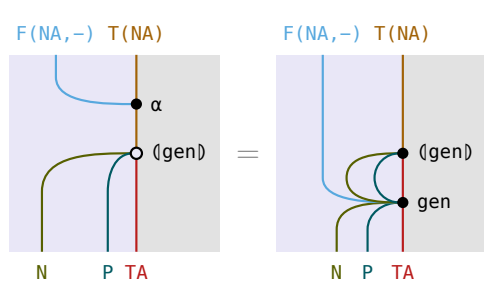
(A.3.3b)

```
gen ≜ F(𝟙,moves trans N(union)) zip N(cp(F(A,–)) E(α))
```

```
u = ((1,2,3,4),({[5]},{[6]},{[7]},{[8]}))      : F(N A,N(E(L A)))
F(𝟙,moves trans N(union))                        𝟙 keeps the column, the path SETS move
  moves({[5]},{[6]},{[7]},{[8]})
   = {({[6]},{[7]},{[8]},{[5]}),
      ({[5]},{[6]},{[7]},{[8]}),
      ({[8]},{[5]},{[6]},{[7]})}                 : E(N(E(L A)))  down, unmoved, up
  trans(that)
   = ({{[6]},{[5]},{[8]}},{{[7]},{[6]},{[5]}},
      {{[8]},{[7]},{[6]}},{{[5]},{[8]},{[7]}}) : N(E(E(L A)))  row k gets rows k-1, k, k+1
  N(union)(that)
   = ({[5],[6],[8]},{[5],[6],[7]},
      {[6],[7],[8]},{[5],[7],[8]})               : N(E(L A))     every path into row k, none dropped
zip(that)
   = ((1,{[5],[6],[8]}),(2,{[5],[6],[7]}),
      (3,{[6],[7],[8]}),(4,{[5],[7],[8]}))       : N(F(A,E(L A)))  each row: its square, and the
                                                                   paths it may be put in front of
N(cp P(α))(that)
   = ({[1,5],[1,6],[1,8]},{[2,5],[2,6],[2,7]},
      {[3,6],[3,7],[3,8]},{[4,5],[4,7],[4,8]}) : N(E(L A))     cp pairs the square with each path,
                                                               α prefixes it: α(1,[5])=[1,5]
```

§A.3.4. gen is an F-algebra; ⦇gen⦈: α⦇gen⦈=F(𝟙,⦇gen⦈)gen

(A.3.4a)



α puts the column back on the list, then the fold reads all of it — the fold reads the rest under F, then one gen puts the column in front

(A.3.4b)

```
xs = [(1,2,3,4),(5,6,7,8)] = α((1,2,3,4),[(5,6,7,8)])   : L(N A)

⦇gen⦈(xs) = gen((1,2,3,4), ⦇gen⦈[(5,6,7,8)])
```

both halves are in gen ≜ F(𝟙,moves trans N(union)) zip N(cp(F(A,–)) E(α)) : the tail folds to ({[5]},{[6]},{[7]},{[8]}), one path per row and each of them one square long, and gen on it is gen ≜ F(𝟙,moves trans N(union)) zip N(cp(F(A,–)) E(α)) 's walk.

§A.3.5. Q : F(N A,N(L A))→N(L A), an F(N A,–)-algebra: Q=F(𝟙,moves trans N(est(R))) zip N(α)

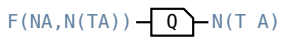
(A.3.5a)

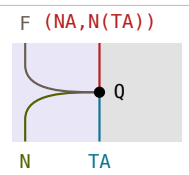
```
Q : F(N A,N(L A)) → N(L A)                    the fold's algebra: a new column, one path per row

u = ((1,2,3,4),([5],[6],[7],[8]))             : F(N A,N(L A))
F(𝟙,moves trans N(est(R)))                       𝟙 keeps the column, the paths move
  moves([5],[6],[7],[8])
   = {([6],[7],[8],[5]),
      ([5],[6],[7],[8]),
      ([8],[5],[6],[7])}                      : E(N(L A))     down, unmoved, up
  trans(that)
   = ({[6],[5],[8]},{[7],[6],[5]},
      {[8],[7],[6]},{[5],[8],[7]})            : N(E(L A))     row k gets rows k-1, k, k+1
  N(est(R))(that)
   = ([5],[5],[6],[5])                        : N(L A)        the cheapest into each row –
                                                              chosen BEFORE the new square
zip(that)
   = ((1,[5]),(2,[5]),(3,[6]),(4,[5]))        : N(F(A,L A))   each row: its square, and the one
                                                              predecessor that survived
N(α)(that)
   = ([1,5],[2,5],[3,6],[4,5])                : N(L A)        α(1,[5])=[1,5]
```

lax(moves) ∧ lax(trans) ∧ lax(zip) ∧ F(𝟙,moves trans N(est(R))) = [l,pm r] ∧ zip = [N(inl),zip' N(inr)] ∧ inl α = wrap ∧ inr α = cons ⟹ Q = [N(wrap),pm zip' N(cons)] (A.3.5b)

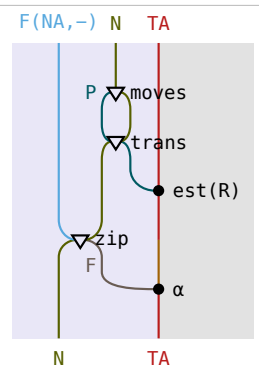
at the last column **Q** starts one path per row, and at each earlier one it puts each square in front of the cheapest of the three kept paths it can step to — the algebra **lax(moves) ∧ lax(trans) ∧ lax(zip) ∧ lax(setify) ∧ R R ⊑ R ∧ F(𝟙,N(est(R))) Q ⊑ gen N(est(R))**
⟹ ⦇Q⦈ setify est(R) ⊑ paths est(R) 's last step folds

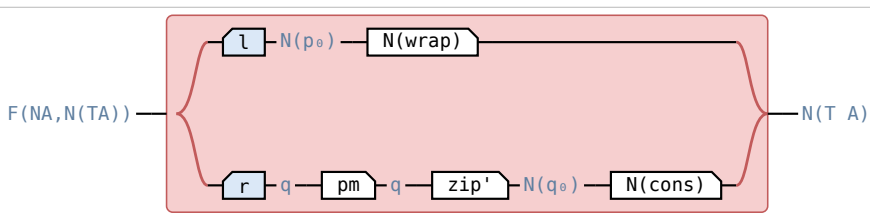
circuit — the fork is F(NA,N(LA))=NA+NA×N(LA) | **Hinze–Marsden**





= the fusion condition lax(moves) ∧ lax(trans) ∧ lax(zip) ∧ R R ⊑ R ∧ F(𝟙,est(R)) α ⊑ cp(F(A,–)) E(α) est(R)
⟹ F(𝟙,N(est(R))) Q ⊑ gen N(est(R)) read as a definition,

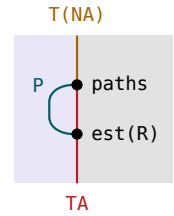
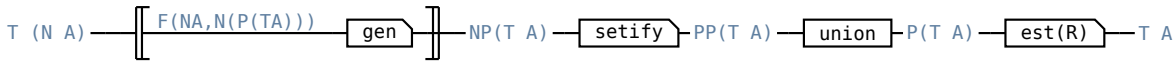
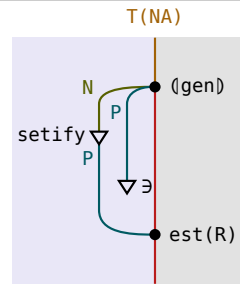
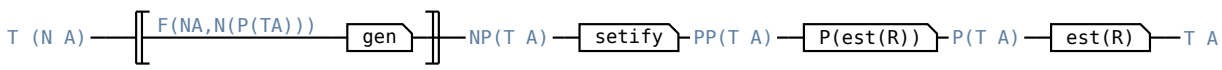
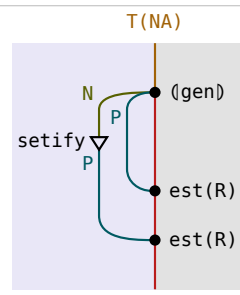
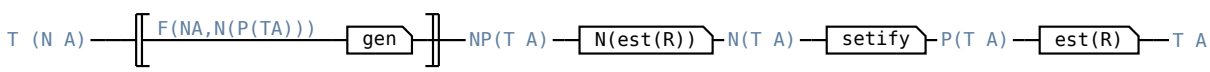
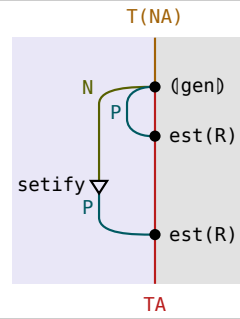
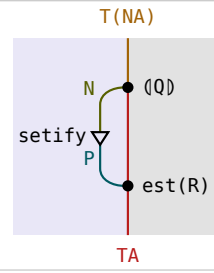


= 

zip=𝟙+zip', α=[wrap,cons]

lax(moves) ∧ lax(trans) ∧ lax(zip) ∧ lax(setify) ∧ R R ⊑ R ∧ F(𝟙,N(est(R))) Q ⊑ gen N(est(R)) ⟹ ⦇Q⦈ setify est(R) ⊑ paths est(R) (A.3.5c)

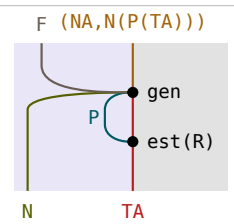
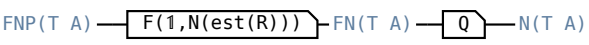
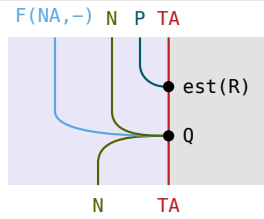
shortest paths on a cylinder: the dynamic programming approach to exhaustive search allows a path of least cost to be found in `O(n×m)` time; `Q` is `lax(moves) ∧ lax(trans) ∧ lax(zip) ∧ F(𝟙,moves trans N(est(R))) = [l,pm r] ∧ zip = [N(inl),zip' N(inr)] ∧ inl α = wrap ∧ inr α = cons ⟹ Q = [N(wrap),pm zip' N(cons)]` 's algebra.

circuit	Hinze–Marsden
	
=  `paths ≜ ⦇gen⦈ setify union` at `paths`	
⊑  `P(est(R)) est(R)⊑union est(R)` — a minimum in each set, then a minimum of those; `R` transitive	
⊑  `setify` lax natural	
⊑ fusion at `lax(moves) ∧ lax(trans) ∧ lax(zip) ∧ R R ⊑ R ∧ F(𝟙,est(R)) α ⊑ cp(F(A,-)) E(α) est(R)` `⟹ F(𝟙,N(est(R))) Q ⊑ gen N(est(R))`	

lax(moves) ∧ lax(trans) ∧ lax(zip) ∧ R R ⊑ R ∧ F(𝟙,est(R)) α ⊑ cp(F(A,−)) E(α) est(R) ⟹ F(𝟙,N(est(R))) Q ⊑ gen N(est(R)) (A.3.5d)

fusion: the condition for fusion in **lax(moves) ∧ lax(trans) ∧ lax(zip) ∧ lax(setify) ∧ R R ⊑ R ∧ F(𝟙,N(est(R))) Q ⊑ gen N(est(R))**

⟹ **⦇Q⦈ setify est(R) ⊑ paths est(R)** 's last step, used to derive a definition of **Q**.

circuit	Hinze–Marsden
	
⊑  (7.13), then zip, trans, moves lax natural	

(7.13) is F(𝟙,est(R))α⊑cp P(α) est(R), Theorem 7.1 Map(f) ∧ 𝟙 ⊑ R ⟹ f° F(R) f ⊑ R ⟺ F(est(R)) f ⊑ $\frac{F(\ni)\ f}{\ni}$ est(R) at the map α with $\frac{F(𝟙,\ni)\alpha}{\ni}$ =cp P(α): extending every path in a set and then taking a minimum is beaten by extending one minimum. It is the crux here, not the greedy theorem.