Companion to Bird & de Moor, Algebra of Programming

§3. Applications

§3.1. Laws from Relation Algebra

(3.1a)

F⊣G	F's type	monad FG	𝟙⊑FG	GF⊑𝟙	F preserves ∪	G preserves ∩	FGF=F	GFG=G
◁⊣▷	A→ A⊗A	◁▷=𝟙	𝟙⊑◁▷	▷◁⊑𝟙	(R ∪ S)◁= R◁ ∪ S◁	(R∩S)▷= R▷∩S▷	◁▷◁=◁	▷◁▷=▷
⊸⊣⊸	A→𝕀	⊸⊸=T	𝟙⊑⊸⊸	⊸⊸⊑𝟙	(R ∪ S)⊸= R⊸ ∪ S⊸	(R∩S)⊸= R⊸∩S⊸	⊸⊸⊸=⊸	⊸⊸⊸=⊸
°⊣°	(A→B)→ (B→A)	R°°=R	R°°=R	R°°=R	(R ∪ S)°=S° ∪ R°	(R∩S)°=R°∩S°	R°°=R	R°°=R
⊸◁⊣▷⊸	(X⊗A→Y)→ (X→Y⊗A)	𝟙	(⊸◁⊗𝟙) (𝟙⊗▷⊸)=𝟙	(𝟙⊗⊸◁) (▷⊸⊗𝟙)=𝟙	=	=	=	=
▷⊸⊣⊸◁	(X→Y⊗A)→ (X⊗A→Y)	𝟙	(𝟙⊗⊸◁) (▷⊸⊗𝟙)=𝟙	(⊸◁⊗𝟙) (𝟙⊗▷⊸)=𝟙	=	=	=	=
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
∪⊣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
𝟘⊣!	{*}→ (A→B)			𝟘⊑R				
·S⊣/S	(A→B)→ (A→C)	S/S	R⊑(RS)/S	(T/S)S⊑T	(S ∪ T)R=SR ∪ TR	(R∩R')/S=(R/S)∩(R'/S)	((RS)/S)S =RS	((T/S)S)/S =T/S
S·⊣S\	(B→C)→ (A→C)	S\S	R⊑S\(SR)	S(S\T)⊑T	R(S ∪ T)=RS ∪ RT	S\(R∩R')= (S\R)∩(S\R')	S(S\(SR)) =SR	S\(S(S\T)) =S\T
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
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
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
·f⊣·f°	(A→B)→ (A→C)	ff°	Map(f) ⟹ 𝟙⊑ ff°	f°f⊑𝟙	(S ∪ T)R=SR ∪ TR	Simple(g) ⟹ (X∩Y)g°= Xg°∩Yg°	ff°f=f	f°ff°=f°
f°·⊣f·	(A→C)→ (B→C)	ff°	Map(f) ⟹ 𝟙⊑ ff°	f°f⊑𝟙	R(S ∪ T)=RS ∪ RT	f(X∩Y)= fX∩fY	ff°f=f	f°ff°=f°
i⊣E	Map↪Rel	E	$\frac{\mathbb{1}}{\ni}$:A→EA	∋:EB→B	—	—	$\frac{\ni}{\ni}$=𝟙	$\frac{R}{\ni}$∋=R
·∋⊣ $\overline{\ni}$	Map(A,EB)→ (A→B)	𝟙	$\frac{f\ni}{\ni}$=f	$\frac{R}{\ni}$∋=R	—	—	$\frac{\frac{f\ni}{\ni}\ni}{\ni}$ =f∋	$\frac{\frac{R}{\ni}\ni}{\ni}$ =$\frac{R}{\ni}$
$\overline{\ni}$ ⊣·∋	(A→B)→ Map(A,EB)	𝟙	$\frac{R}{\ni}$∋=R	$\frac{f\ni}{\ni}$=f	—	—	$\frac{\frac{R}{\ni}\ni}{\ni}$ =$\frac{R}{\ni}$	$\frac{\frac{f\ni}{\ni}\ni}{\ni}$ =f∋

(3.1b)

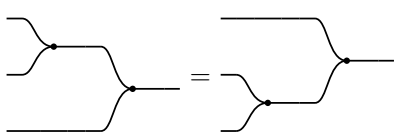
	·T	T·	·g	g°·	T∩	°	⊸◁	Δ
·S	R/S₁ S₂=R/S₂/S₁	S\(R/W)=(S\R)/W	R/(Sg)=(Rg°)/S	Map(f) ⟹ f(R/S)=fR/S	—	(Y/S)°=S°\(Y°)	(RS)^=R^(S⊗𝟙)	(R∩R')/S=(R/S)∩(R'/S)
S·	S\(R/W)=(S\R)/W	S₁ S₂\R=S₂\(S₁\R)	S\(Rg°)=(S\R)g°	(g°S)\R=S\(gR)	—	(S\Y)°=Y°/S°	((S⊗𝟙)R)^=SR^	S\(R∩R')=(S\R)∩(S\R')
·f	Map(f) ⟹ R/fS=(R/S)f°	T\(Rf°)=(T\R)f°	(RS)°=S°R°			(RS)°=S°R°		Simple(g) ⟹ (X∩Y)g°=Xg°∩Yg°
f°·	Map(f) ⟹ f(R/S)=fR/S	(Tf°)\R=f(T\R)	—	(RS)°=S°R°		(RS)°=S°R°		f(T₁∩T₂)=fT₁∩fT₂
R∩	—	—	—	—	R⇨S⇨T=R∩S⇨T	(R⇨Y)°=R°⇨(Y°)	—	R⇨S∩T=(R⇨S)∩(R⇨T)
°	(Y/T)°=T°\(Y°)	(T\Y)°=Y°/T°	(RS)°=S°R°	(RS)°=S°R°	(T⇨Y)°=T°⇨(Y°)	R°°=R	(R^)°=(R°⊗𝟙)(𝟙⊗▷⊸) both bends: the ° section's display	(T₁∩T₂)°=T₁°∩T₂°
⊸◁	(RT)^=R^(T⊗𝟙)	((T⊗𝟙)R)^=TR^	—	—	—	(R^)°=(R°⊗𝟙)(𝟙⊗▷⊸) both bends: the ° section's display	—	—
∪	(S ∪ T)R=SR ∪ TR	R(S ∪ T)=RS ∪ RT	(S ∪ T)R=SR ∪ TR	R(S ∪ T)=RS ∪ RT	R∩(S ∪ T)=R∩S ∪ R∩T	(R ∪ S)°=S° ∪ R°		
𝟘	𝟘R=𝟘	R𝟘=𝟘	𝟘R=𝟘	R𝟘=𝟘	R∩𝟘=𝟘	𝟘°=𝟘		

(3.1c)

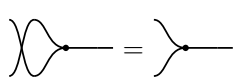
·f⊣·f°⊣/f°

f°·⊣f·⊣f\

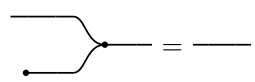
(3.1d)



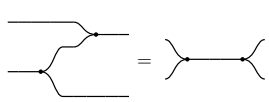
▷ associative



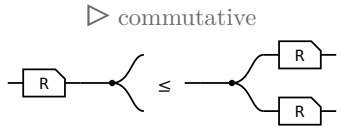
▷ commutative



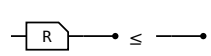
⊸ is its unit



Frobenius, one half — the other is its °



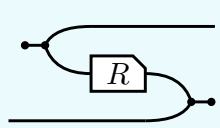
R◁≤◁(R⊗R)



R⊸≤⊸

(3.1e)

° 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:

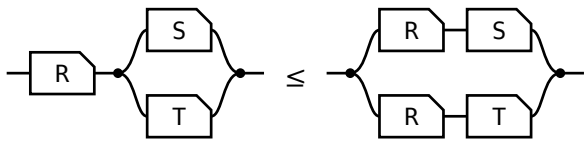


R°=(⊸◁⊗𝟙)(𝟙⊗R⊗𝟙)(𝟙⊗▷⊸)

where ⊸◁ : 𝕀→a⊗a opens a pair of wires out of nothing and ▷⊸ : b⊗b→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: ◁ and ⊸ are ▷° and ⊸°. The laws of ° are that it is a contravariant 2-functor ° : $\mathcal{C}^{op}$→$\mathcal{C}$:

(i) 𝟙°=𝟙 (ii) (RS)°=S°R° (iii) (R⊗S)°=R°⊗S° (iv) R≤S implies R°≤S°

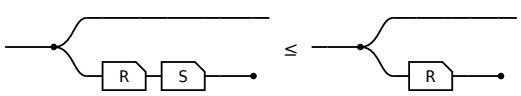
(3.1f)

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.	

(3.1g)

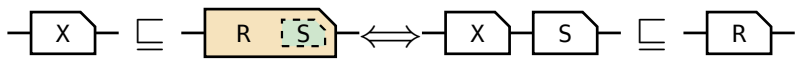
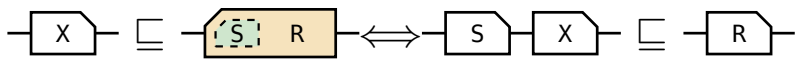
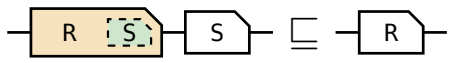
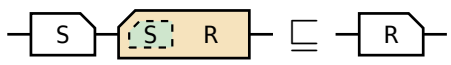
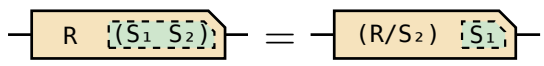
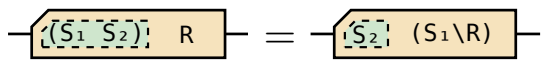
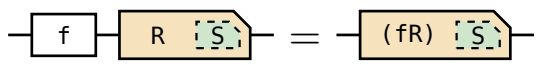
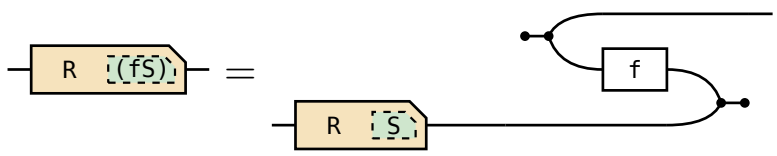
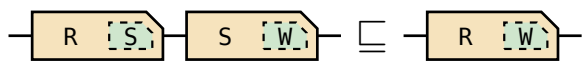
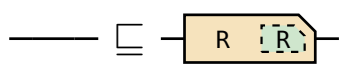
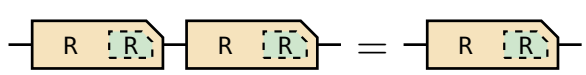
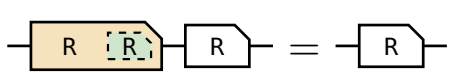
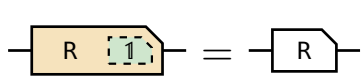
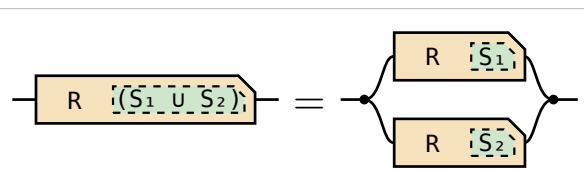
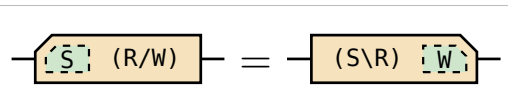
dom(R)≜𝟙∩RR°
Coreflexive(X) ⟹ dom(R)⊑X ⟺ R⊑XR
dom(RS)⊑dom(R)
dom(R∩S)=𝟙∩SR°
R entire Entire(R)≜dom(R)=𝟙
R simple Simple(R)≜R°R⊑𝟙
R a map Map(R)≜Entire(R) ∧ Simple(R)
Entire(R) ∧ Entire(S) ⟹ Entire(RS) Simple(f) ∧ Simple(g) ⟹ Simple(fg) Map(f) ∧ Map(g) ⟹ Map(fg)
Entire(RS) ⟹ Entire(R)

(3.1h)



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

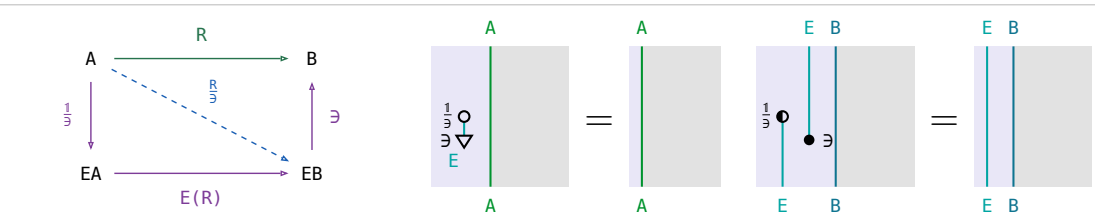
(3.1i)

$X \sqsubseteq R/S \Longleftrightarrow XS \sqsubseteq R$ 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.	
$X \sqsubseteq S\backslash R \Longleftrightarrow SX \sqsubseteq R$ The mirror — divide on the left when x comes first.	
$(R/S)S \sqsubseteq R$ 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, $(R/S)S=\varnothing$.	
$S\ (S\backslash R) \sqsubseteq R$ The mirror.	
associate: $R/(S_1S_2)=(R/S_2)/S_1$ **A friend's enemy** is two hops: divide by the far end first.	
$(S_1S_2)\backslash R=S_2\backslash(S_1\backslash R)$ The mirror.	
maps: $f\ (R/S)=(fR)/S$ Rename x before or after dividing — the licence to write fR/S.	
$R/(fS)=(R/S)f^\circ$ Rename y: a map leaves a denominator as f° outside the box.	
$(R/S)(S/W) \sqsubseteq R/W$ Someone who admires all of a hate-set that already covers everyone z works for admires those people too.	
$\mathbb{1} \sqsubseteq R/R$ 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.	
$(R/R)(R/R)=R/R$ R/R is the preorder **admires at least as much as**, and a preorder is idempotent. Freyd writes $\sqsubseteq$; with $\mathbb{1} \sqsubseteq R/R$ above it is an equality.	
$(R/R)R=R$ Reaching p through someone whose idols x fully admires is reaching p directly, since x admires their own idols.	
$R/\mathbb{1}=R$ Dividing by $\mathbb{1}$: p's set is just $\{p\}$, so admiring all of it is admiring p.	
$R/(S_1 \cup S_2)=R/S_1 \cap R/S_2$ Admiring a combined hate-set is admiring each set in full.	
$S\backslash(R/W)=(S\backslash R)/W$ Which is why $S\backslash R/W$ needs no bracket.	

(3.1j)

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→EB, 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: $\frac{1}{\ni}$ ($\frac{1}{\ni}$)°⊑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) ⟹ $\frac{fR}{\ni}$=f$\frac{R}{\ni}$	Naturality of the unit, Map(f) ⟹ f$\frac{1}{\ni}$=$\frac{1}{\ni}$ 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$\frac{1}{\ni}$	Rename first or take singletons first — the fusion row above at R=𝟙.
11 R□=∋□ ⟹ R/ₛS=$\frac{R}{\ni}$ ($\frac{S}{\ni}$)°	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): EA→EB, image of a set of A
13 $\frac{R}{\ni}$=$\frac{1}{\ni}$ E(R)	$\frac{1}{\ni}$: x↦{x}
14 $\frac{S}{\ni}$ E(R)=$\frac{SR}{\ni}$	absorption — the monad's composition law, $\frac{S}{\ni}$ ⋄ $\frac{R}{\ni}$ = $\frac{SR}{\ni}$, §$\mathcal{A}$≅Kleisli(E) in Relation Algebra
15 ⊇≜∋/∋	xs⊇ys⟺∀a. ys∋a→xs∋a
16 ⊆=⊇°	xs⊆ys⟺ys⊇xs

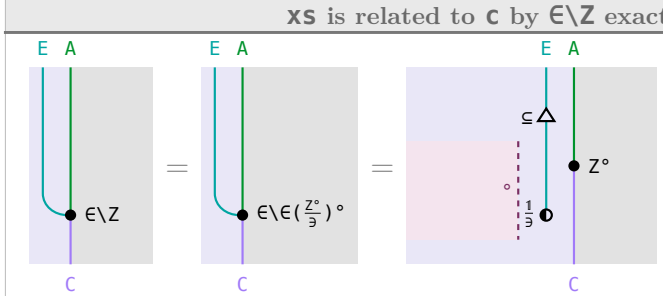
(3.1k)



$\frac{R}{\ni}$∋=R EA is the powerset of A — standard mathematics, but here P is already the relator P(R).

∈\Z=⊆($\frac{Z°}{\ni}$)° (3.1l)

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



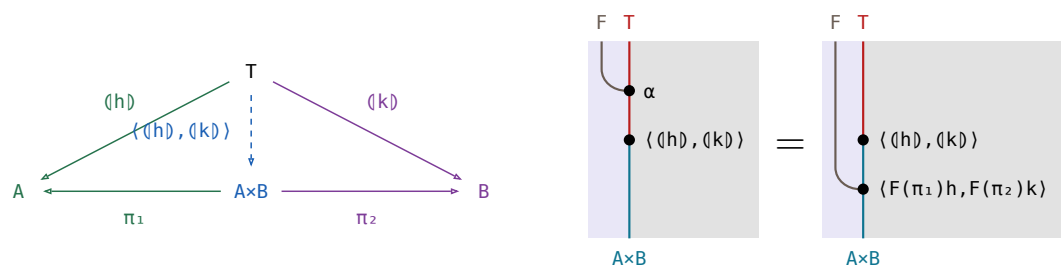
(a) (b)

(a) Z=∈Λ(Z°)° — (3.1j)
(b) X\(Yf°)=(X\Y)f°, f a map

(3.1m)

definition	**type**	**note**
`listr A::=nil|cons(A,listr A)`	$\mathcal{A}\to\mathcal{A}$	The cons-lists over `A`, the datatype every row below folds.
`sum=⦇[zero,plus]⦈`	`[A]→A`	`plus(a,b)=a+b`.
`length=⦇[zero,π₂ succ]⦈`	`[A]→Nat`	`π₂` drops the head and keeps the count of the tail, `succ` adds one for the head.
`average≜⟨sum,length⟩div`	`[Nat]→Nat`	`div(m,n)=m/n`, with `div(0,0)=0` so `average` is total. Traverses the list twice.
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.
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.
the instance `⟨sum,length⟩=⦇[zeros,pluss]⦈`	`[Nat]→Nat×Nat`	`pluss(a,(b,n))=(a+b,n+1)`, so `average` runs in one pass.
`preds n=[n,n−1,…,1]`	`Nat→[Nat]`	apply the earlier special case to write `preds` as `⦇k⦈π₁`.

(3.1n)

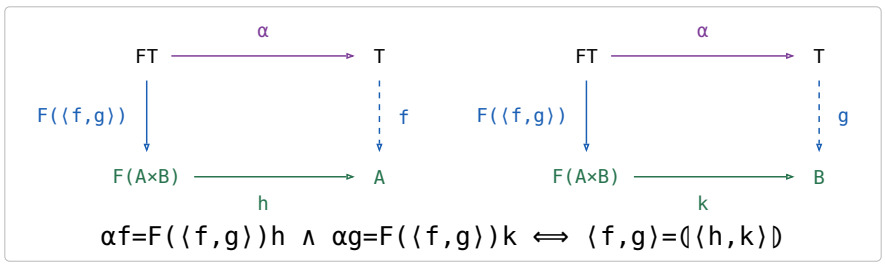


`⟨⦇h⦈,⦇k⦈⟩=⦇⟨F(π₁)h,F(π₂)k⟩⦈` banana split

§3.2. Fokkinga's mutual recursion theorem

- `F-Alg(𝒜)` — the algebras and their homomorphisms — has binary products whenever 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 (3.1m) 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**.

(3.2a)



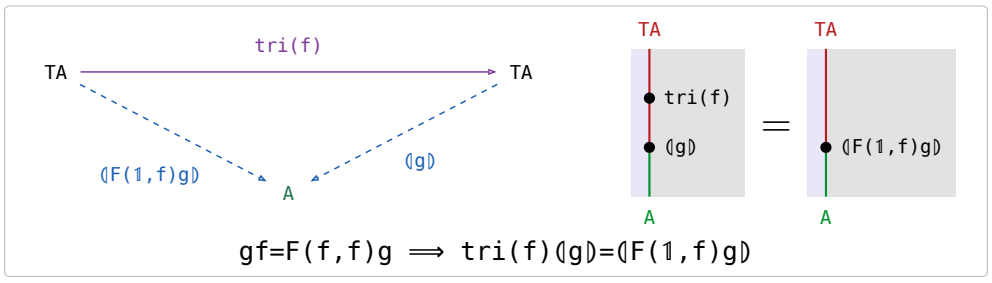
§3.3. Ruby triangles

(3.3a)

stage	definition
informally	`tri(f)[a₀,a₁,…,aᵢ,…,aₙ]=[a₀,f(a₁),…,fⁱ(aᵢ),…,fⁿ(aₙ)]`
for cons-lists	`tri(f)=⦇[nil,(𝟙×listr(f)) cons]⦈`
with the base functor named	`F(A,B)=1+A×B`, `α=[nil,cons]`, so `tri(f)=⦇F(𝟙,listr(f))α⦈`
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`.

(3.3b)



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`.

§3.4. Depth of a tree

(3.4a)

the datatype	`tree A::=tip(A)|bin(tree A,tree A)`, base functor `F(A,B)=A+B×B`, initial type `([tip,bin],tree)`
the fold	`⦇[g,h]⦈` is the unique `f` with `f(tip(a))=g(a)` and `f(bin(x,y))=h(f(x),f(y))`
`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))`
`max`	`max=⦇[𝟙,bmax]⦈`, where `bmax(a,b)` is the larger of `a` and `b`
`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
`depth`	`depth=depths max`

§5. Datatypes in Allegories

§5.1. Relators

(5.1a)

Every hom-set of an allegory is a poset, so an allegory is a **locally posetal 2-category**: the 2-cell from R to S IS R⊑S. A **relator** F : $\mathcal{C}\to\mathcal{D}$ is a 2-functor between allegories:

$$F(\mathbb{1})=\mathbb{1} \qquad F(RS)=F(R)F(S) \qquad R\sqsubseteq S \implies F(R)\sqsubseteq F(S)$$

Preserving ° is **not** asked for — ° is an identity-on-objects involution $\mathcal{C}^{op}\to\mathcal{C}$, no part of the 2-category.

(5.1b)

the statement
Map(f) ⟹ Map(F(f)) and Map(f) ⟹ F(f°)=F(f)° .
Over a **tabular** allegory a functor is a relator ⟺ it preserves °.
F(R°)=F(R)° for every R, so F(R)° needs no bracket.
Two relators agreeing on maps are equal.
Coreflexive(X) ∧ Coreflexive(Y) ⟹ F(X∩Y)=F(X)∩F(Y)
F(R∩S)⊑F(R)∩F(S) , and strictly.
PreservesRecip(F) ⟹ F(dom(R))=dom(F(R))

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. Fork ⟨R,S⟩

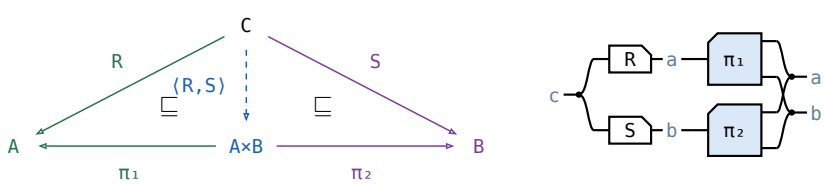
(5.2a)

The **fork** of R : C→A and S : C→B is ⟨R,S⟩≜Rπ₁°∩Sπ₂° , where (π₁,π₂) is the tabulation of ⊤ .

(5.2b)

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

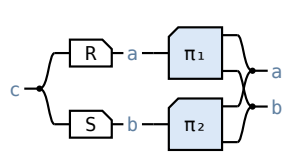
(5.2c)



A domain is coreflexive, so ⟨R,S⟩π₁⊑R, with equality exactly when S is entire; for maps both triangles commute and ⟨f,g⟩ is unique. In Rel, c ⟨R,S⟩ (a,b) iff c R a and c S b — copy c, then R on one strand and S on the other, which is ◁(R⊗S) on the right.

No ° survives the translation. π₁=𝟙⊗⊸ discards the second component, so π₁°=𝟙⊗⊷ **creates** one out of nothing, and ∩ 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:

(5.2d)



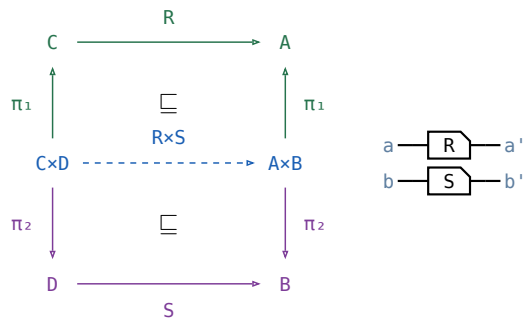
⊷▷=𝟙 on each half

§5.2.1. Relational product R×S

(5.2.1a)

R×S≜⟨π₁R,π₂S⟩ , a relator in each argument but no longer a categorical product.

(5.2.1b)

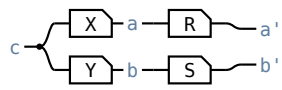


Right-then-up is `(R×S)π₁`, up-then-right is `π₁R`, and `(R×S)π₁⊑π₁R`, equality when `S` is entire. In `Rel`, `(c,d) (R×S) (a,b)` iff `c R a` and `d S b` — two strands side by side, no copy dot: `R×S=R⊗S`.

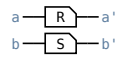
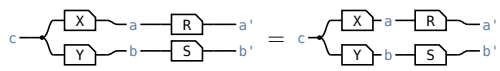
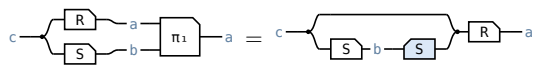
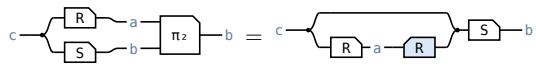
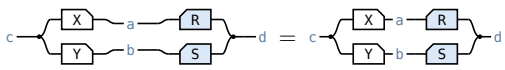
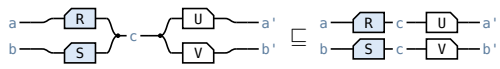
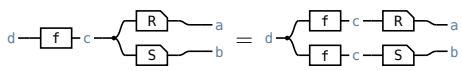
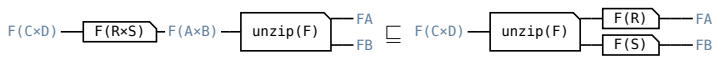
§5.2.2. Absorption

For `X : E→C` and `Y : E→D`, `⟨X,Y⟩(R×S)=⟨XR,YS⟩`. Both sides are this picture:

(5.2.2a)

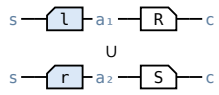
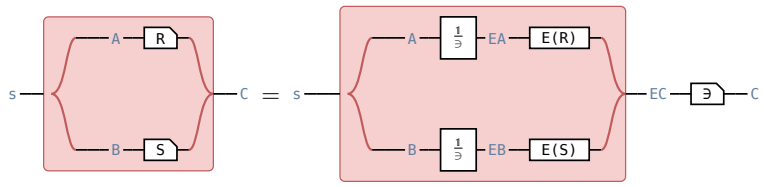
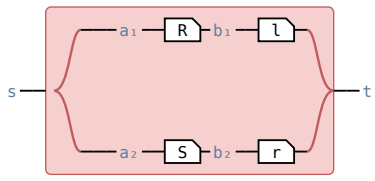
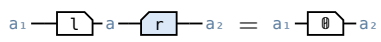
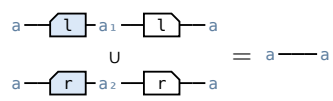
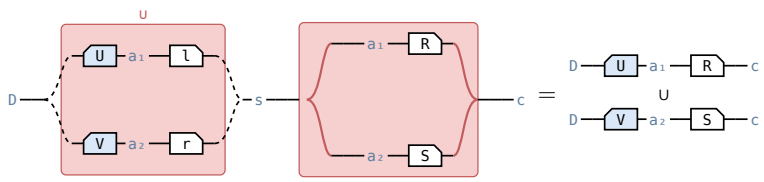


(5.2.2b)

	the law	picture
1	`R×S≜⟨π₁ R,π₂ S⟩`	
2	`⟨X,Y⟩(R×S)=⟨XR,YS⟩` both sides are the same strokes — absorption. Its ⊑ half is that half at `S:=𝟙` and at `R:=𝟙` — stages of their proof of this row; the corollary is the `(R×S)(U×V)=(RU)×(SV)` it yields, at `R:=𝟙` and `V:=𝟙`.	
3	`⟨R,S⟩π₁=dom(S)R` (5.2b)	
4	`⟨R,S⟩π₂=dom(R)S` (5.2b)	
5	`⟨X,Y⟩⟨R,S⟩°=XR°∩YS°`	
6	`⟨R,S⟩°⟨U,V⟩⊑(R°U)×(S°V)`	
7	`Map(f) ⟹ f⟨R,S⟩=⟨fR,fS⟩` `f` a map; it fails for an arbitrary arrow;	
8	`F(R×S)unzip(F)⊑unzip(F)(F(R)×F(S))` `unzip(F)≜⟨F(π₁),F(π₂)⟩`; only ⊑ for an arbitrary `R`, `S`	
9	`g=curry(f)⟺(g×𝟙)eval=f` reading × as the relational product, does `Rel` have exponentials?	

§5.3. Coproduct `[R,S] : A+B→C`

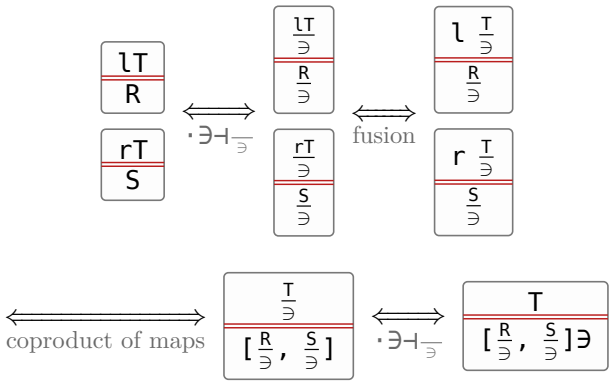
(5.3a)

the statement	picture
`[R,S]≜l°R ∪ r°S` 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.	
`[R,S]=[R/∋,S/∋]∋`	
`R+S≜[Rl,Sr]`	
`l[R,S]=R , r[R,S]=S` `lT=R ∧ rT=S ⟹ T=[R,S]` ,	
`ll°=𝟙 , rr°=𝟙`	
`lr°=⓪ , rl°=⓪`	
`l°l ∪ r°r=𝟙`	
`[U,V]°[R,S]=U°R ∪ V°S`	

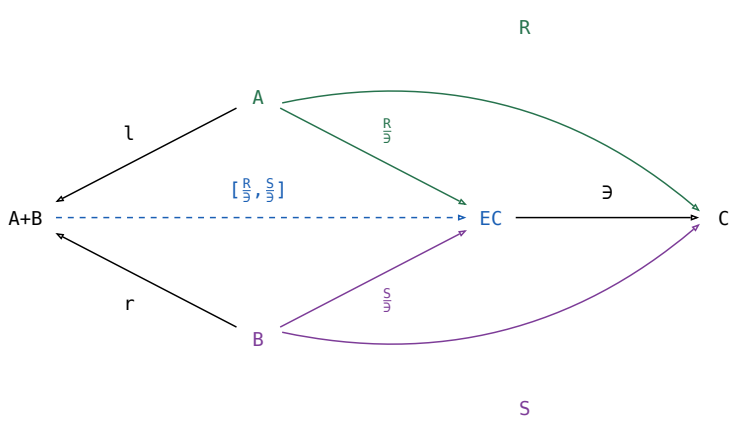
§5.3.1. `[R,S]≜[R/∋, S/∋]∋`

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→EC` reversibly, so the map coproduct can be applied underneath it. For any `T : A+B→C`,

(5.3.1a)

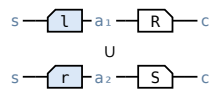
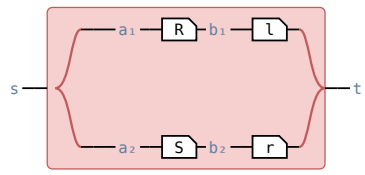
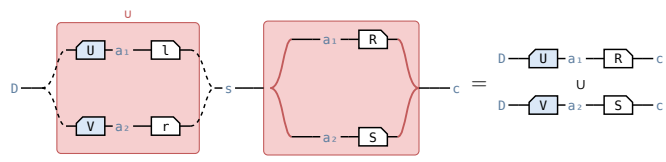


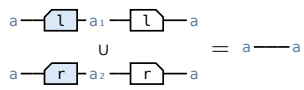
(5.3.1b)



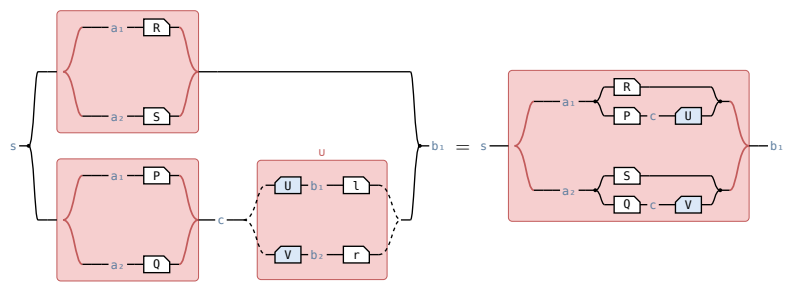
Nothing here holds only up to ⊑: every triangle commutes on the nose, which is the difference from the fork above. The border spells `[R,S]=[`$\frac{R}{\ni}$`, `$\frac{S}{\ni}$`]∋`, and pushing ∋ into the union that `[·,·]` on maps already is turns that back into the definition, `(l°`$\frac{R}{\ni}$`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

(5.3.1c)

the law	**picture**
`[R,S]≜l°R ∪ r°S` (5.3a)'s first row	
`R+S≜[Rl,Sr]`	
`[U,V]°[R,S]=U°R ∪ V°S` (5.3a)'s last row;	

`[𝟙,𝟘]=l°` , `[𝟘,𝟙]=r°`
`[l,r]=𝟙`
which is (5.9)
	

prove (5.11), and say why duality does not carry it over from the product law	

`(R+S)∩[P,Q][U,V]°=`
`(R∩PU°)+(S∩QV°)`
`[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)

For R : A→B,

P(R)≜(∈\R∈)∩(∋R/∋)

E(R)≜ $\frac{\ni R}{\ni}$ = ((∋R)/∋)∩(∋/(∋R))°

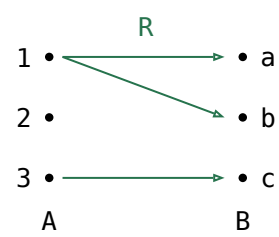
xs P(R) ys⟺(∀a∈xs. ∃b∈ys. a R b)∧(∀b∈ys. ∃a∈xs. a R b)

Every element of xs is related by R to some element of ys, and conversely.

(5.4b)

	in Rel	in words	im(xs)	partner
(∋R)/∋	∀y∈ys. ∃x∈xs. x R y	∀y. some xs R y	ys⊆im(xs)	every y has a partner
(∋/(∋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
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
((∋R°)/∋)°	∀x∈xs. ∃y∈ys. x R y	∀x. x R some ys	—	every x has a partner
P(R)	∀y∈ys. ∃x∈xs. x R y and ∀x∈xs. ∃y∈ys. x R 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)
∅	∅	all 8 subsets of abc	∅	all 8 subsets of abc	∅
{1}	∅, a, b, ab	ab, abc	ab	a, b, ab, ac, bc, abc	a, b, ab
{2}	∅	all 8 subsets of abc	∅	none	none
{3}	∅, c	c, ac, bc, abc	c	c, ac, bc, abc	c
{1,2}	∅, a, b, ab	ab, abc	ab	none	none
{1,3}	all 8 subsets of abc	abc	abc	ac, bc, abc	ac, bc, abc
{2,3}	∅, c	c, ac, bc, abc	c	none	none
{1,2,3}	all 8 subsets of abc	abc	abc	none	none

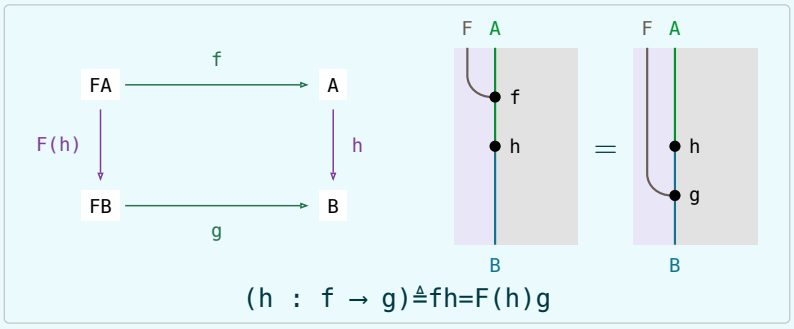
(5.4d)

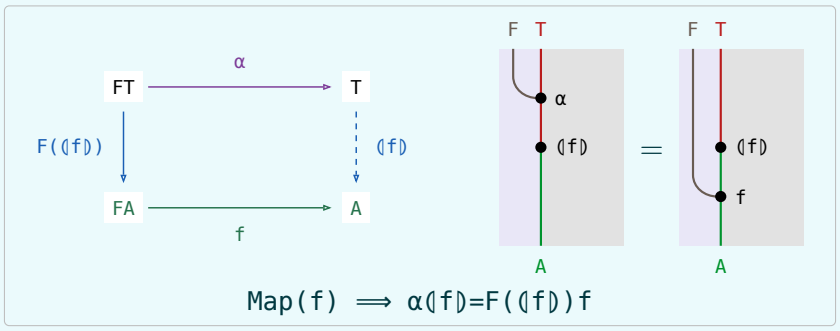
law	the reading
X⊑P(R)⟺X∋⊑∋R and X°∋⊑∋R°	One containment, and the same one at R° — which is the definition read off the two divisions. Hence P(R°)=P(R)°, and R⊑S⟹P(R)⊑P(S) .
P(𝟙)=𝟙	The straightness axiom verbatim: extensionality **is** P's unit law.
Map(f) ⟹ P(f)=E(f)	In Rel, xs P(f) ys⟺ys={f(a)|a∈xs}. 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.
P(RS)=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).

§5.5. Initial algebra

(5.5a)

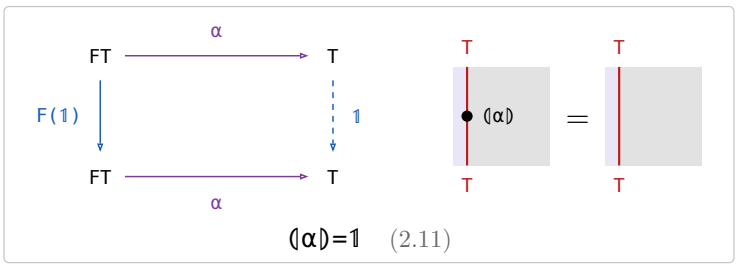
An **F-algebra** is a map `f : F(A)→A`; `A` is its **carrier**. An **F-homomorphism** from `f : F(A)→A` to `g : F(B)→B` is a map `h : A→B` with `f h=F(h)g`. The **initial algebra** `α : F(T)→T` is the F-algebra with exactly one F-homomorphism `⦇f⦈ : T→A` to every F-algebra `f` .





§5.5.1. Reflection

(5.5.1a)

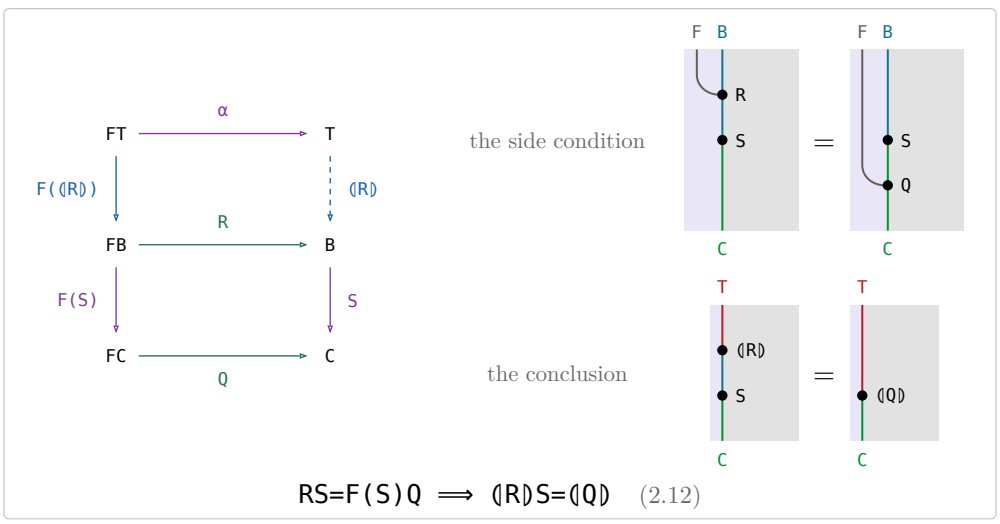


Taking a value apart with α and putting it straight back is doing nothing.

§5.5.2. Fusion: ⦇R:FB⦈ absorb S:R->Q (in Alg(F)) and becomes ⦇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`.

(5.5.2a)



§5.6. Type relator

(5.6a)

Let F be a binary relator with initial type (α,T), so T is a type functor. F(R,S) is its action on a pair, and F(X) abbreviates F(𝟙,X), the F of the reduce section. For every object A the initial algebra is α : F(A,TA)→TA, among the maps. T acts on an arrow R : A→B by

$$T(R)=(\!|F(R,\mathbb{1})\alpha|\!)$$

(5.6b)

name	law	what it says
the defining equation	T(R)=(|F(R,𝟙)α|)	Rebuild the structure with α, applying R to the parameter on the way.
functor	T(𝟙)=𝟙 and T(R)T(S)=T(RS)	Acting by the identity changes nothing, and two actions in a row are one action.
type functor fusion	T(f)(|h|)=(|F(f,𝟙)h|)	A relator action followed by a fold is a single fold — the intermediate structure is never built. The side condition holds because F is a bifunctor — F(R,𝟙)F(𝟙,(|Q|))=F(R,(|Q|))=F(𝟙,(|Q|))F(R,𝟙).
naturality of α	αT(f)=F(f,T(f))α	Building and then mapping is the same as mapping the parts and then building, so α is natural from G(R)=F(R,T(R)) to T.
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.

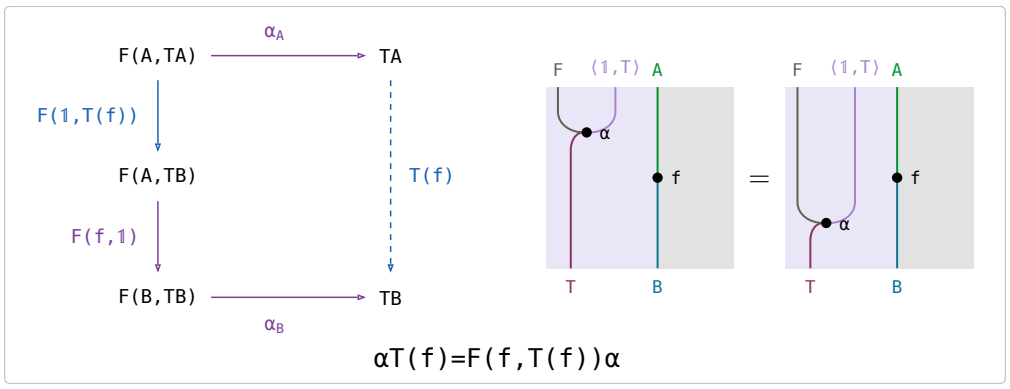
§5.6.1. Type functor

(5.6.1a)

Let F be a bifunctor taking both the parameter A and the recursive position TA, with an initial algebra α_A : F(A,TA)→TA for every object A. Then T is a functor, acting on a map f : A→B by

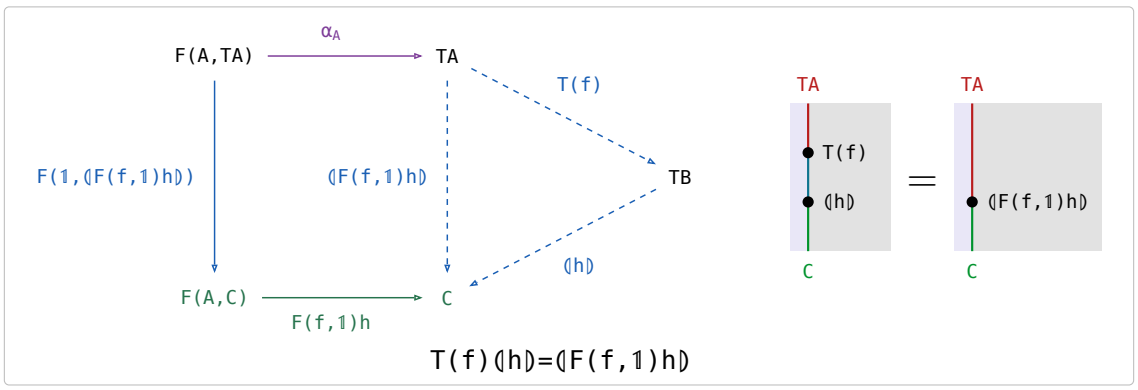
$$T(R) \triangleq (\!| F(R,\mathbb{1})\alpha |\!)$$

(5.6.1b)



- 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: ⟨𝟙,T⟩ : $\mathcal{A}\to\mathcal{A}\times\mathcal{A}$ sends A to (A,TA), and F(⟨𝟙,T⟩(A)) is F(A,TA).
- The picture is three wires — F, ⟨𝟙,T⟩, and the object — and the region between the first two is $\mathcal{A}\times\mathcal{A}$.
- α is then an ordinary natural transformation F∘⟨𝟙,T⟩⇒T: its bead eats the F and ⟨𝟙,T⟩ wires, and the T wire is born under it.
- F(f,T(f)) costs no notation. It is the bead f on the object wire with ⟨𝟙,T⟩ and F running past: ⟨𝟙,T⟩ is what turns f into the pair (f,T(f)), and F is what applies it.
- The law is the naturality of α, which is exactly the freedom to slide that f bead past it.

(5.6.1c)



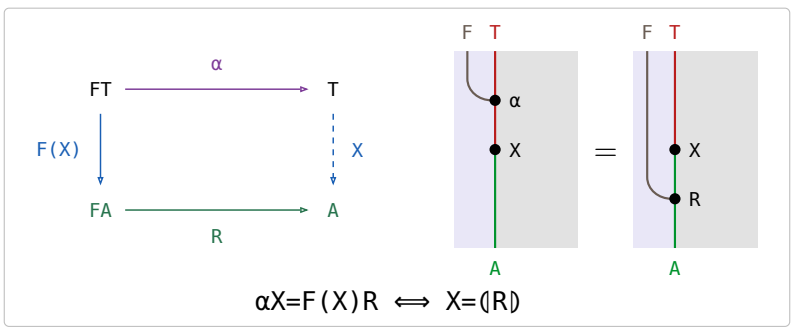
§5.7. Reduce

(5.7a)

let F be a relator and has **initial algebra** α : F(T)→T in the subcategory of functions. α is also initial in the allegory:

§5.7.1. The defining equation

(5.7.1a)



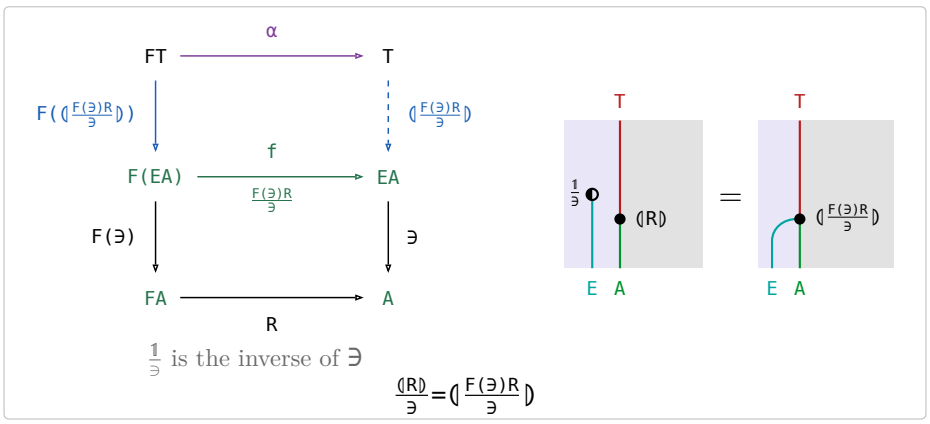
An F-algebra is a **weakened** natural transformation. A transformation F⇒𝟙 on $\mathcal{A}$ would need a component FX→X at every object and a commuting square at every arrow, but F-Algebra only need it works on T and A.

(5.7.1b)

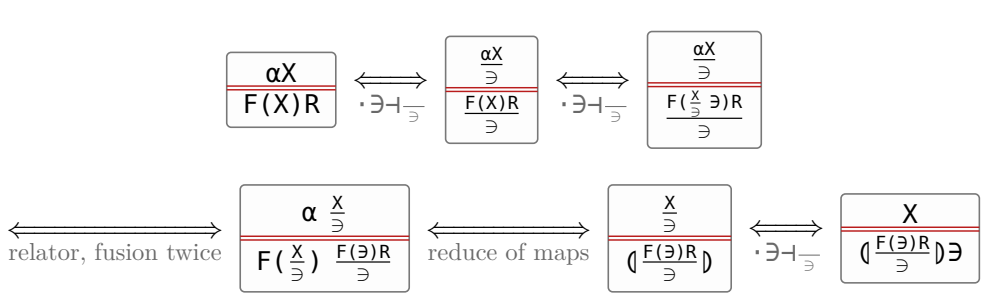
⦇[c,f]⦈=reduce(c,f) catamorphism: arrows of the form ⦇f⦈ are called catamorphisms, and these two examples reveal the notion to be a familiar idea in abstract clothing		
part	**Nat**	**[A]**
datatype	Nat::=zero|succ Nat	[A]::=nil|cons(A,[A])
base functor F	F(X)=1+X	F(X)=1+A×X
initial algebra α	α=[zero,succ]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

§5.7.2. ⦇R⦈=⦇$\frac{F(\ni)R}{\ni}$⦈∋

(5.7.2a)



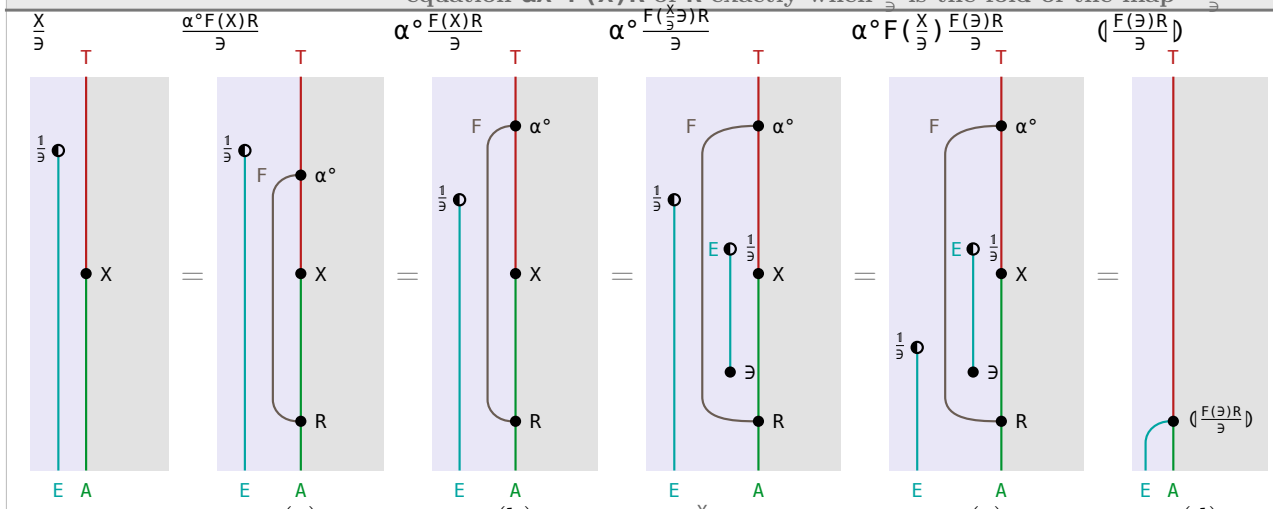
(5.7.2b)



the last two rows at X:=⦇R⦈:

$$\alpha X = F(X)R \iff \frac{X}{\ni} = (\!|\frac{F(\ni)R}{\ni}|\!) \qquad (5.7.2c)$$

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 $\frac{F(\ni)R}{\ni}$



(a) α iso; $\alpha X = F(X)R$
(b) α° is a map because α is an iso, so it leaves Λ: $\frac{\alpha^\circ F(X)R}{\ni} = \alpha^\circ \frac{F(X)R}{\ni}$ — (3.1j)
(c) $F(\frac{X}{\ni}\ni) = F(\frac{X}{\ni})F(\ni)$ 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, (3.1j)
(d) fold uniqueness, $\frac{F(\ni)R}{\ni}$ being a map — (5.5a)

§5.8. Combinatorial functions

(5.8a)

definition	type	note
`[A]::=nil|cons(A,[A])`	$\mathcal{A}\to\mathcal{A}$	The list type, under the short name it keeps.
`list(R)=(|[nil,(R×𝟙)cons]|)`	`[A]→[B]`	The relator's action on `R : A→B`: one `R` per element, the shape untouched.
`subseq=(|[nil,cons ∪ π₂]|)`	`[A]→[A]`	`xs subseq ys`: `ys` is `xs` with elements dropped — `cons` keeps the head, `π₂` drops it.
`prefix=(|[nil,⊸ nil ∪ cons]|)` `prefix=cat°π₁ =init*`	`[A]→[A]`	`ys` is an initial segment of `xs`; the first `nil` is where it stops early. `init≜snoc° π₁`.
`suffix=cat°π₂ =tail*`	`[A]→[A]`	The dual, `tail≜cons° π₂`; as a reduce it needs snoc-lists.
`segment=suffix prefix`	`[A]→[A]`	A contiguous stretch of `xs`: a suffix, then a prefix of that.
`partition=concat°`	`[A]→[[A]]`	This `cat` is restricted to `[A]⁺×[A]→[A]`, so `ys` is a list of non-empty segments of `xs`.
`concat=(|[nil,cat]|)`	`[[A]]→[A]`	Joins the segments back up, which is why its converse splits a list.
`inits`	`[A]→[[A]]`	Implements $\frac{\texttt{prefix}}{\ni}$, listing the prefixes by increasing length.
`tails`	`[A]→[[A]]`	Implements $\frac{\texttt{suffix}}{\ni}$ by decreasing length — the opposite order.
`filter(p)≜` $\frac{\texttt{subseq list(p)}}{\ni}$ `est(R°)`	`[A]→[A]`	The longest subsequence of `xs` whose every element passes `p`. `est(R°)` is ≜ `est`
`R≜length≤length°`	`[A]→[A]`	The preorder `filter` and `takewhile` maximise over: the longer list wins. `≥≜≤°`
`takewhile(p)≜` $\frac{\texttt{prefix list(p)}}{\ni}$ `est(R°)`	`[A]→[A]`	The same with `prefix` for `subseq`: the longest prefix whose every element passes `p`.
`mss≜` $\frac{\texttt{segment sum}}{\ni}$ `est(≥)`	`[A]→A`	Maximum segment sum. `segment=suffix prefix` splits it into $\frac{\texttt{prefix sum}}{\ni}$ `est(≥)` on each suffix.

§5.8.1. $\frac{\texttt{subseq}}{\ni}$ =(|[nil $\frac{\mathbb{1}}{\ni}$,($\frac{\mathbb{1}\times\ni}{\ni}$ E(cons),π₂) cup]|)

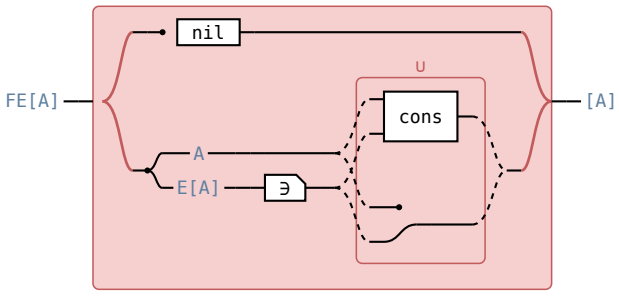
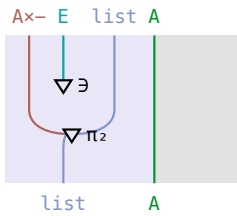
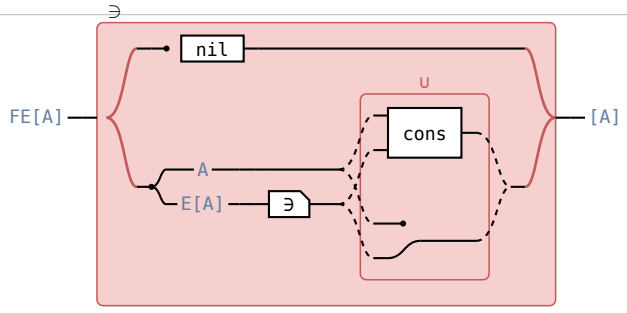
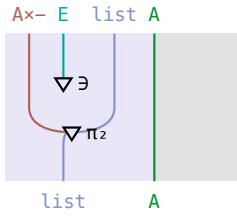
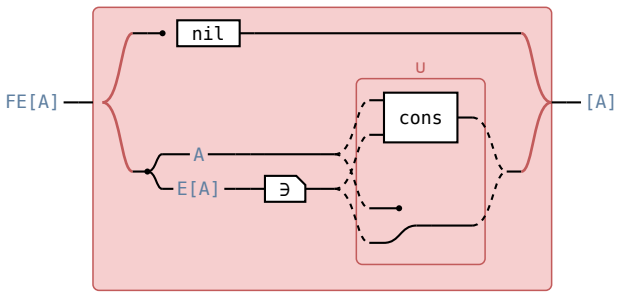
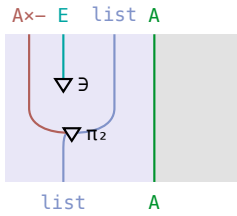
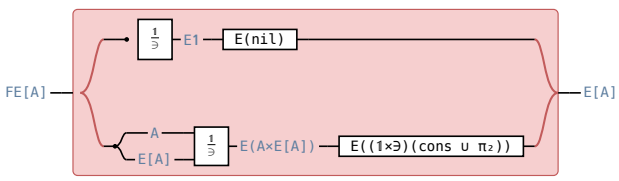
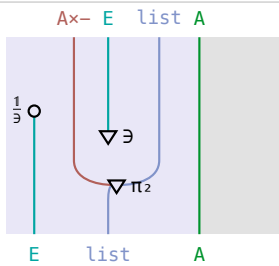
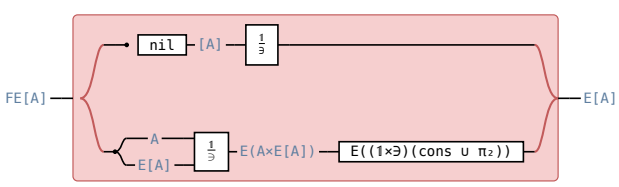
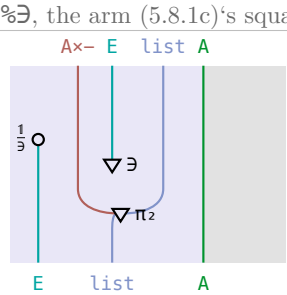
(5.8.1a)

$\mathsf{cup} \triangleq \frac{\pi_1 \ni \cup \pi_2 \ni}{\ni} : \mathsf{EA \times EA \to EA}$, so $\frac{R \cup S}{\ni} = \langle \frac{R}{\ni}, \frac{S}{\ni} \rangle\ \mathsf{cup}$.

(5.8.1b)

$$\frac{F(\ni)[nil, cons \cup \pi_2]}{\ni} = [nil\ \tfrac{1}{\ni}, \tfrac{(1 \times \ni)(cons \cup \pi_2)}{\ni}]$$

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 — (5.7.2b) at $subseq = (\!|[nil, cons \cup \pi_2]|\!)$, $\triangleq$ **subseq**.

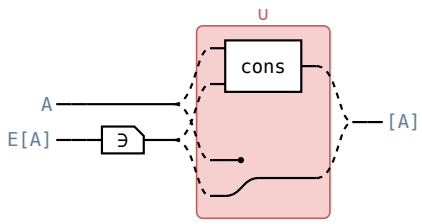
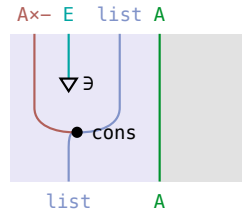
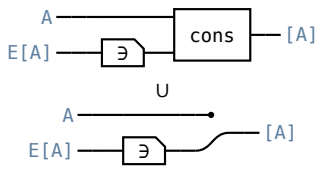
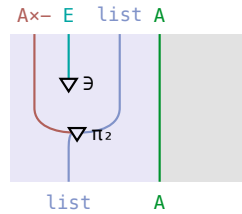
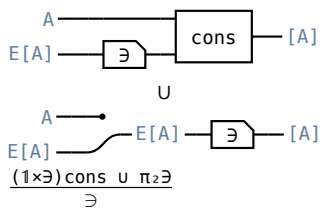
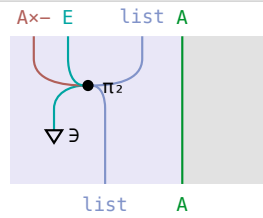
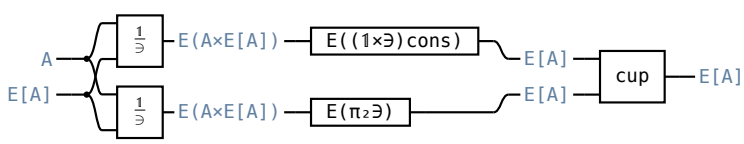
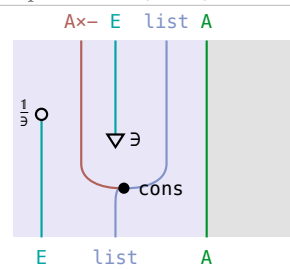
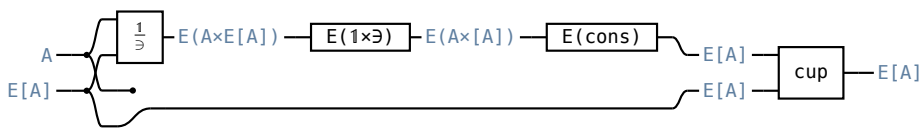
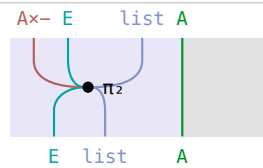
circuit — the fork is $F([A]) = 1 + A \times [A]$: nil above, the pair below	**Hinze–Marsden**
 $\frac{F(\ni)[nil, cons \cup \pi_2]}{\ni}$	 the π_2 operand of $cons \cup \pi_2$ under the $1 \times \ni$ summand of $F(\ni)$, i.e. $(1 \times \ni)\pi_2$
=  $\frac{(1+1\times\ni)[nil, cons \cup \pi_2]}{\ni}$ $F(X) = 1 + A \times X$ — $\triangleq$ subseq	 the same operand under $1+1\times\ni$, whose $1\times\ni$ summand it sits in
=  $\frac{[nil, (1\times\ni)(cons \cup \pi_2)]}{\ni}$ $R + S \triangleq [Rl, Sr]$, $l[R,S] = R$, $r[R,S] = S$ — (5.3a)	 the π_2 operand of the second arm $(1 \times \ni)(cons \cup \pi_2)$
=  $[\frac{nil}{\ni}, \frac{(1\times\ni)(cons \cup \pi_2)}{\ni}]$ (5.3.1a) at $T := [nil, (1\times\ni)(cons \cup \pi_2)]$	 the π_2 operand under its $1\%\ni$, the arm (5.8.1c)'s square rewrites, $(1\times\ni)\pi_2 = \pi_2\ni$
=  $[nil\ \tfrac{1}{\ni}, \tfrac{(1\times\ni)(cons \cup \pi_2)}{\ni}]$ (3.1j), $\frac{f}{\ni} = f\ \frac{1}{\ni}$ for f a map, at $f := nil$	 the same operand; the two rows differ only in the nil arm

(5.8.1c)

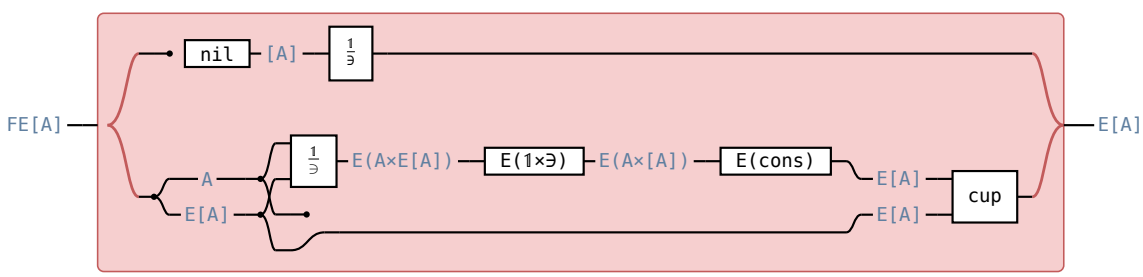
$$\begin{tikzcd}
A \times E[A] \arrow[r, "\pi_2"] \arrow[d, "1\times\ni"'] & E[A] \arrow[d, "\ni"] \\
A \times [A] \arrow[r, "\pi_2"'] & {[A]}
\end{tikzcd}$$

$$\frac{(\mathbb{1}\times\ni)(\mathtt{cons}\ \cup\ \pi_2)}{\ni} = \langle \frac{\mathbb{1}\times\ni}{\ni}\ \mathtt{E(cons)}, \pi_2 \rangle\ \mathtt{cup} \qquad (5.8.1d)$$

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
 $\frac{(\mathbb{1}\times\ni)(\mathtt{cons}\ \cup\ \pi_2)}{\ni}$ (5.8.1b)'s second branch	 the cons operand of $\mathtt{cons}\ \cup\ \pi_2$
=  $\frac{(\mathbb{1}\times\ni)\mathtt{cons}\ \cup\ (\mathbb{1}\times\ni)\pi_2}{\ni}$ $T(X_1\ \cup\ X_2)=TX_1\ \cup\ TX_2$ — (3.1b)	 the π_2 operand of $(\mathbb{1}\times\ni)\mathtt{cons}\ \cup\ (\mathbb{1}\times\ni)\pi_2$
=  $\frac{(\mathbb{1}\times\ni)\mathtt{cons}\ \cup\ \pi_2\ni}{\ni}$ $(\mathbb{1}\times\ni)\pi_2=\pi_2\ni$ — (5.2.1b) at π_2, an equality because $\mathbb{1}$ is entire	 the π_2 operand of $(\mathbb{1}\times\ni)\mathtt{cons}\ \cup\ \pi_2\ni$
=  $\langle \frac{(\mathbb{1}\times\ni)\mathtt{cons}}{\ni}, \frac{\pi_2\ni}{\ni} \rangle$ cup $\frac{R\ \cup\ S}{\ni} = \langle \frac{R}{\ni}, \frac{S}{\ni} \rangle$ cup — $\triangleq$ cup	 the cons operand under its $\mathbb{1}\%\ni$
=  $\langle \frac{\mathbb{1}\times\ni}{\ni}\ \mathtt{E(cons)}, \pi_2 \rangle$ cup (3.1j), 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

(5.8.1e)



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

which writes Pcons; cons is a map, and there $P(\mathtt{cons})=E(\mathtt{cons})$ — (5.4d).

§5.9. Lax natural transformations (LaT)

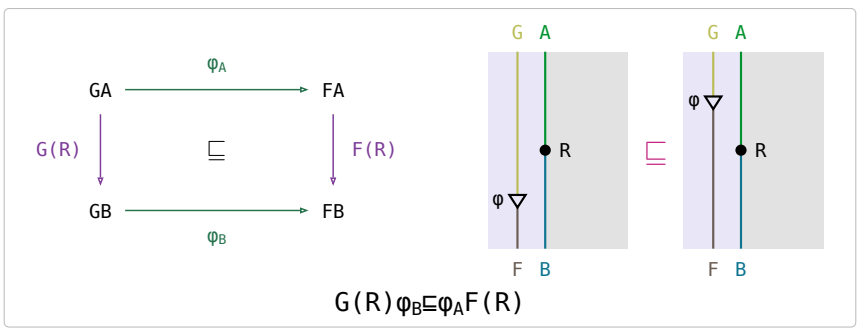
(5.9a)

For relators $G,F : \mathcal{C}\to\mathcal{D}$ and components $\varphi_A : GA\to FA$, φ is **lax at** $R : A\to B$ when $G(R)\varphi_B\sqsubseteq\varphi_A F(R)$ — both sides $GA\to FB$, the left through the component at B, the right through the one at A.

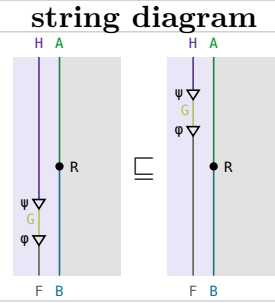
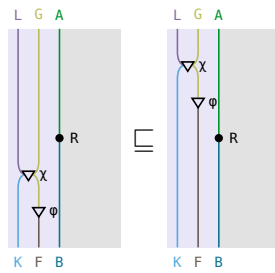
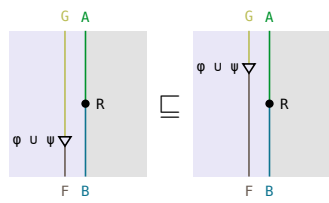
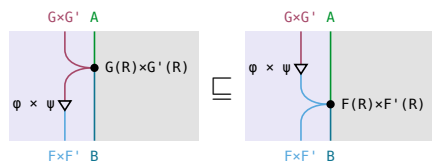
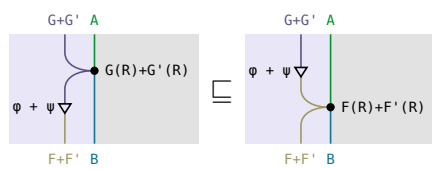
$\varphi : G\Rightarrow F$ is a **lax natural transformation** (LaT) when it is lax at **every** R. (5.13)

Lax at every **map** already gives LaT, and at a map the inequation is an equality $G(f)\varphi=\varphi F(f)$: laxness is about relations only.

(5.9b)



(5.9c)

closed under	commutative diagram	string diagram
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 $H(R)(\psi_B\varphi_B) \sqsubseteq (\psi_A\varphi_A)F(R)$	
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	
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 $G(R)(\varphi_B \cup \psi_B) \sqsubseteq (\varphi_A \cup \psi_A)F(R)$	
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 $K(G(R))K(\varphi_B) \sqsubseteq K(\varphi_A)K(F(R))$	
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 $R\triangleleft \sqsubseteq \triangleleft(R\times R)$ — (3.1d)	
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 cofork is the derived case $[\varphi,\psi] = (\varphi+\psi)\triangledown$, and $\triangledown$ costs nothing	
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 (3.1f) a counterexample: (5.9.1a)	

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

§5.9.1. meet is not closed in LaT

(5.9.1a)

counterexample — $\varphi,\psi : G\Rightarrow F$ lax natural does NOT give $\varphi\cap\psi$ lax natural

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

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

$A=B\triangleq\{0,1\}$, $R\triangleq\{(0,0),(1,0)\}$, $\varphi\triangleq\pi_1\cap\pi_2 : \Delta\Rightarrow Id$

§5.9.2. Two stacked towers

(5.9.2a)

	lower — inside ONE allegory	**upper — between allegories**
0-cell	an object A	an allegory $\mathcal{C}$ — **a whole lower tower**
1-cell	a relation $R : A\to B$	a relator $F : \mathcal{C}\to\mathcal{D}$
2-cell	$R\sqsubseteq S$	a LaT $\varphi : G\Rightarrow F$ $\triangleq$ lax
order	$R\sqsubseteq S$	$\varphi\sqsubseteq\psi$ componentwise
union	$R \cup S$	$\varphi \cup \psi$, **survives** (5.9c)
product	$R\times S$ $\triangleq \times$	$\varphi\times\psi$, **survives**; a TENSOR, not a categorical product; SYMMETRIC (5.2b) (5.9c)
coproduct	$R+S$	$\varphi+\psi$, **survives**; a BIPRODUCT (5.9c)
meet	$R\cap S$	$\varphi\cap\psi$ componentwise, **fails** (5.9.1a)
converse	R°	φ°, **fails**, oplax: $\varphi_A{}^\circ G(R)\sqsubseteq F(R)\varphi_B{}^\circ$
zero object	z with $\mathbb{1}$ $z=\mathbb{0}$	the constant relator at such a z, initial and terminal

§5.9.3. R⊑S is a special case of LaT

(5.9.3a)

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

$G\triangleq$const A, $F\triangleq$const B — the relators $X\mapsto A$ and $X\mapsto B$, each sending every arrow to $\mathbb{1}$ $\triangleq$ lax

with $\mathcal{C}$ two objects and one non-identity arrow $X\to Y$, a LaT const $A\Rightarrow$const B is the pair $\varphi_X,\varphi_Y : A\to B$ with $\varphi_Y\sqsubseteq\varphi_X$ — the datum $R\sqsubseteq S$

not every LaT is constant: $\ni : P\Rightarrow Id$ (5.4d) $\pi_2 : (A\times-)\Rightarrow Id$ (5.8.1c) $\lhd : Id\Rightarrow\Delta$ (3.1d)

§6. Recursive Programs

§6.1. φ(Y)⊑Y⟹(μX : φ(X))⊑Y

(6.1a)

φ a **monotonic** mapping of the hom-set A→B into itself: X⊑Y⟹φ(X)⊑φ(Y) .

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

(6.1b)

the law	what it says
φ(Y)⊑Y ⟹ (μX : φ(X))⊑Y	to bound (μX : φ(X)) above, exhibit one Y the body does not grow past — the half §6.2 and every chapter after it uses
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

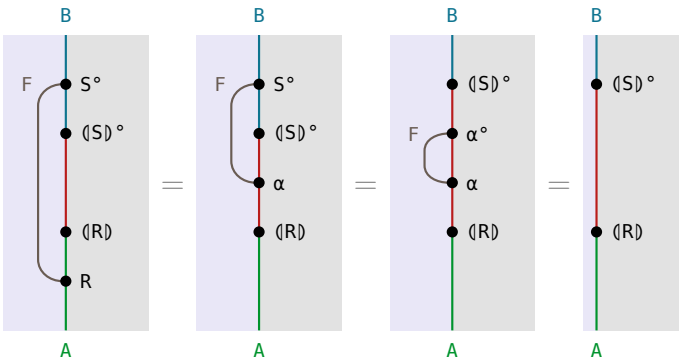
§6.2. ⦇S⦈°⦇R⦈=(μX : S°F(X)R)

⦇S⦈°⦇R⦈=(μX : S°F(X)R) (6.2a)

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

S°F(⦇S⦈°⦇R⦈)R=⦇S⦈°⦇R⦈

hylomorphism theorem: a prototypical 'divide and conquer' scheme — the term S° represents the decomposition stage, F(⦇S⦈°⦇R⦈) the stage of solving the subproblems recursively, and R the recombination stage; R : FA→A, S : FB→B, α : FT→T initial

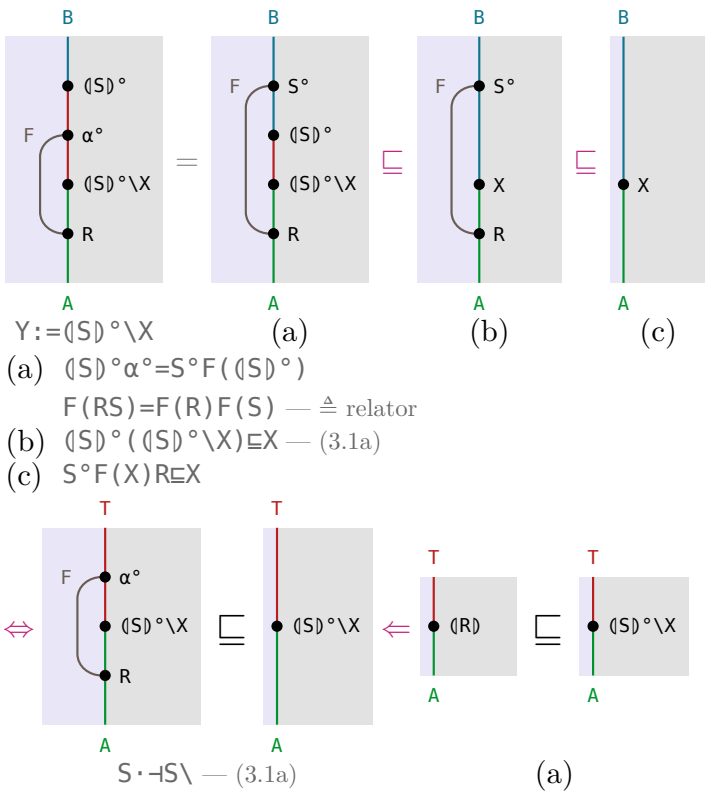


(a) (b) (c) α iso

(a) the body at ⦇S⦈°⦇R⦈
F(RS)=F(R)F(S) — ≜ relator
(b) F(⦇R⦈)R=α⦇R⦈ — ≜ reduce
(c) ⦇S⦈°α°=S°F(⦇S⦈°) — ≜ reduce, (5.1b)

S°F(X)R⊑X ⟹ ⦇S⦈°⦇R⦈⊑X

hylomorphism theorem: by Knaster–Tarski, the hylomorphism ⦇S⦈°⦇R⦈ is included in X if X satisfies the associated recursion inequation



Y:=⦇S⦈°\X (a) (b) (c)

(a) ⦇S⦈°α°=S°F(⦇S⦈°)
F(RS)=F(R)F(S) — ≜ relator
(b) ⦇S⦈°(⦇S⦈°\X)⊑X — (3.1a)
(c) S°F(X)R⊑X

S·⊣S\ — (3.1a) (a)

(a) ⦇R⦈=(μX : α°F(X)R) — ≜ reduce, (6.1b), (3.1a)

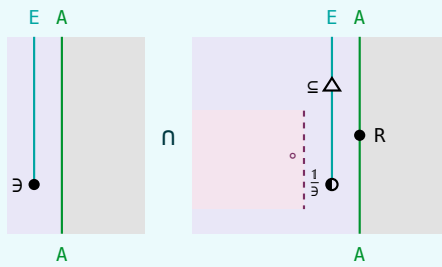
§7. Optimisation Problems

§7.1. est(R)≜∋∩(∈\R°)

(7.1a)

For R : A→A, est(R)≜∋∩(∈\R°) : EA→A X⊑∈\R°⟺∈X⊑R°, and ∈X runs y→xs→x, member first, so its pair (y,x) is in R° exactly when x R y; at R≜≤ that is x≤y for every y∈xs, the least — ∈\≤ would give y≤x, the greatest.

est(R)=∋∩⊆(R/∋)° x is the est(R) of xs when x∈xs and xs ⊆ Λ(R)(x), the set of every y with x R y — (3.1l) at Z≜R°



xs (est(R)) x⟺x∈xs∧(∀y∈xs. x R y) the same predicate under the same letter, min R, so est(R)=min(R°) once R is an arrow;

xs (est(R)) x⟺(x in xs) and all x R\: xs in q

est(R)=∋∩all R° all R≜∈\R, q's all; the chains below keep it written ∈\

E(R)≜ ∋R/∋ : EA→EB, xs E(R) ys⟺ys={y|∃x∈xs. x R y} the image of xs, (3.1j)

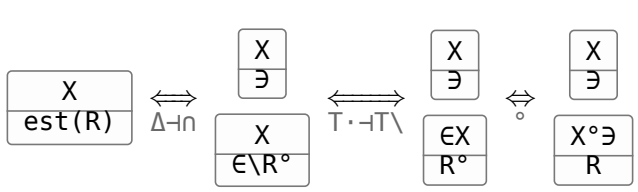
P(R) : EA→EB, xs P(R) ys⟺(∀x∈xs. ∃y∈ys. x R y)∧(∀y∈ys. ∃x∈xs. x R y) every x and every y has a partner, (5.4b)

(7.1b)

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

§7.1.1. X⊑est(R)⟺X⊑∋ and X°∋⊑R

(7.1.1a)



§7.1.2. 1/∋ (∈\R)=R

(7.1.2a)

$$\begin{array}{c}X\\ \hline \frac{1}{\ni}\ (\in\backslash R)\end{array} \xleftrightarrow{f^\circ\cdot\dashv f\cdot} \begin{array}{c}\frac{1}{\ni}{}^\circ X\\ \hline \in\backslash R\end{array} \xleftrightarrow{T\cdot\dashv T\backslash} \begin{array}{c}\in\ \frac{1}{\ni}{}^\circ X\\ \hline R\end{array} \overset{\circ}{\Leftrightarrow} \begin{array}{c}(\frac{1}{\ni}\ \ni)^\circ X\\ \hline R\end{array} \xleftrightarrow{i\dashv E} \begin{array}{c}X\\ \hline R\end{array}$$

§7.1.3. $\frac{S}{\ni}$ (∈\R)=S°\R

$\frac{S}{\ni}$ 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.

(7.1.3a)

$$\begin{array}{c}X\\ \hline \frac{S}{\ni}\ (\in\backslash R)\end{array} \xleftrightarrow{f^\circ\cdot\dashv f\cdot} \begin{array}{c}\frac{S}{\ni}{}^\circ X\\ \hline \in\backslash R\end{array} \xleftrightarrow{T\cdot\dashv T\backslash} \begin{array}{c}\in\ \frac{S}{\ni}{}^\circ X\\ \hline R\end{array} \overset{\circ}{\Leftrightarrow} \begin{array}{c}(\frac{S}{\ni}\ \ni)^\circ X\\ \hline R\end{array} \xleftrightarrow{\cdot\ni\dashv\frac{}{\ni}} \begin{array}{c}S^\circ X\\ \hline R\end{array} \xleftrightarrow{T\cdot\dashv T\backslash} \begin{array}{c}X\\ \hline S^\circ\backslash R\end{array}$$

§7.1.4. union (∈\R)=∈\(∈\R)

(7.1.4a)

$$\begin{array}{c}X\\ \hline \text{union}\ (\in\backslash R)\end{array} \xleftrightarrow{f^\circ\cdot\dashv f\cdot} \begin{array}{c}\text{union}^\circ\ X\\ \hline \in\backslash R\end{array} \xleftrightarrow{T\cdot\dashv T\backslash} \begin{array}{c}\in\text{union}^\circ\ X\\ \hline R\end{array} \overset{\circ}{\Leftrightarrow} \begin{array}{c}(\text{union}\ni)^\circ X\\ \hline R\end{array} \xleftrightarrow{\cdot\ni\dashv\frac{}{\ni}} \begin{array}{c}(\ni\ni)^\circ X\\ \hline R\end{array} \xleftrightarrow{\circ,\ T\cdot\dashv T\backslash} \begin{array}{c}X\\ \hline \in\backslash(\in\backslash R)\end{array}$$

§7.1.5. $\frac{S}{\ni}$ est(R)=S∩(S°\R°)

(7.1.5a)

$$\begin{array}{c}X\\ \hline \frac{S}{\ni}\ \text{est}(R)\end{array} \xleftrightarrow{f^\circ\cdot\dashv f\cdot} \begin{array}{c}\frac{S}{\ni}{}^\circ X\\ \hline \text{est}(R)\end{array} \xleftrightarrow{\Delta\dashv\cap} \begin{array}{c}\frac{S}{\ni}{}^\circ X\\ \hline \ni\end{array}\ \begin{array}{c}\frac{S}{\ni}{}^\circ X\\ \hline \in\backslash R^\circ\end{array} \xleftrightarrow{f^\circ\cdot\dashv f\cdot} \begin{array}{c}X\\ \hline \frac{S}{\ni}\ \ni\end{array}\ \begin{array}{c}X\\ \hline \frac{S}{\ni}\ (\in\backslash R^\circ)\end{array} \xleftrightarrow{\cdot\ni\dashv\frac{}{\ni},\ (7.1.3a)} \begin{array}{c}X\\ \hline S\end{array}\ \begin{array}{c}X\\ \hline S^\circ\backslash R^\circ\end{array} \xleftrightarrow{\Delta\dashv\cap} \begin{array}{c}X\\ \hline S\cap(S^\circ\backslash R^\circ)\end{array}$$

§7.1.6. $\frac{S}{\ni}$ est(R)= $\frac{S}{\ni}$ est(R∩S°S)

(7.1.6a)

$$\begin{array}{c}X\\ \hline \frac{S}{\ni}\ \text{est}(R\cap S^\circ S)\end{array} \xleftrightarrow{(7.1.5a),\ \circ} \begin{array}{c}X\\ \hline S\cap(S^\circ\backslash(R^\circ\cap S^\circ S))\end{array} \xleftrightarrow{\Delta\dashv\cap,\ T\cdot\dashv T\backslash} \begin{array}{c}X\\ \hline S\end{array}\ \begin{array}{c}S^\circ X\\ \hline R^\circ\cap S^\circ S\end{array}$$

$$\xleftrightarrow{\Delta\dashv\cap,\ S^\circ\cdot\ \text{monotone}} \begin{array}{c}X\\ \hline S\end{array}\ \begin{array}{c}S^\circ X\\ \hline R^\circ\end{array} \xleftrightarrow{T\cdot\dashv T\backslash,\ \Delta\dashv\cap} \begin{array}{c}X\\ \hline S\cap(S^\circ\backslash R^\circ)\end{array} \xleftrightarrow{(7.1.5a)} \begin{array}{c}X\\ \hline \frac{S}{\ni}\ \text{est}(R)\end{array}$$

§7.1.7. P(f) est(R)=est(fRf°) f

(7.1.7a)

$$\begin{array}{c}P(f)\ \text{est}(R)\\ \text{f a map}\end{array} \xlongequal{\text{P=E on maps},\ (7.1.5a)} (\ni f)\cap((\ni f)^\circ\backslash R^\circ) \xlongequal{\circ,\ T\cdot\dashv T\backslash,\ f^\circ\cdot\dashv f\cdot} (\ni f)\cap(\in\backslash(fR^\circ))$$

$$\xlongequal{\text{modular law, f simple}} (\ni\cap((\in\backslash(fR^\circ))f^\circ))f \xlongequal{\cdot f\dashv\cdot f^\circ,\ \circ,\ \text{est}} \text{est}(fRf^\circ)\ f$$

§7.1.8. P(S) est(R)⊑(∋S)∩(∈\(SR°))

(7.1.8a)

$$\begin{array}{c}P(S)\ \text{est}(R)\\ \hline (\ni S)\cap(\in\backslash(SR^\circ))\end{array} \xleftarrow{\Delta\dashv\cap,\ T\cdot\dashv T\backslash} \begin{array}{c}P(S)\ \text{est}(R)\\ \hline \ni S\end{array}\ \begin{array}{c}\in P(S)\ \text{est}(R)\\ \hline SR^\circ\end{array} \xleftarrow{\text{UP of est}} \begin{array}{c}P(S)\ni\\ \hline \ni S\end{array}\ \begin{array}{c}\in P(S)\\ \hline S\in\end{array}$$

§7.1.9. P(est(R)) est(R)⊑union est(R)

(7.1.9a)

$$\begin{array}{c}\texttt{P(est(R)) est(R)}\\ \hline \texttt{union est(R)}\end{array} \underset{\text{R transitive}}{\overset{(7.1.5a)}{\Longleftrightarrow}} \begin{array}{c}\texttt{P(est(R)) est(R)}\\ \hline \texttt{(∋∋)∩((∋∋)°\R°)}\end{array} \xleftarrow{\;°,\ \Delta\dashv\cap,\ \mathsf{T}\cdot\dashv\mathsf{T}\backslash\;} \begin{array}{c}\texttt{P(est(R)) est(R)}\\ \hline \texttt{∋∋}\end{array}$$

$$\begin{array}{c}\texttt{∈∈P(est(R))}\\ \texttt{est(R)}\\ \hline \texttt{R°}\end{array} \xleftarrow{\;\text{UP of }\texttt{est}\text{, R transitive}\;} \begin{array}{c}\texttt{P(est(R))∋}\\ \hline \texttt{∋est(R)}\end{array} \quad \begin{array}{c}\texttt{∈P(est(R))}\\ \hline \texttt{est(R)∈}\end{array}$$

, R transitive

§7.2. Monotonic algebras

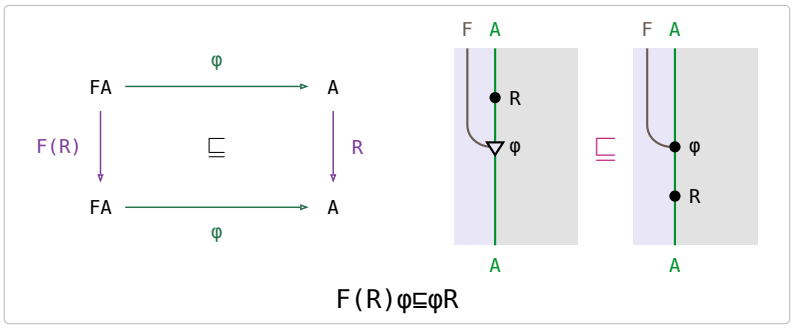
(7.2a)

An F-algebra $\varphi : FA \to A$ is **monotonic on** $R : A \to A$ when it is lax at R at $G:=F$, $F:=Id$: $F(R)\varphi \sqsubseteq \varphi R$. R is an **endorelation**, so $B=A$ and the two components φ_A, φ_B are the one arrow φ — an algebra, not a family.

For a map $f : FA \to A$ that is $f^\circ F(R) f \sqsubseteq R$ (3.1a)'s $f^\circ \cdot \dashv f \cdot$ at $X:=F(R)f$, $Y:=R$, equivalently $F(R) \sqsubseteq f R f^\circ \cdot f \dashv \cdot f^\circ$ then $f^\circ \cdot \dashv f \cdot$, .

$(\leq \times \leq) + \sqsubseteq + \leq$ — addition on Nat is monotonic on $\leq$, which at the point level reads $c=a+b \wedge a \leq a' \wedge b \leq b' \Longrightarrow c \leq a'+b'$.

(7.2b)



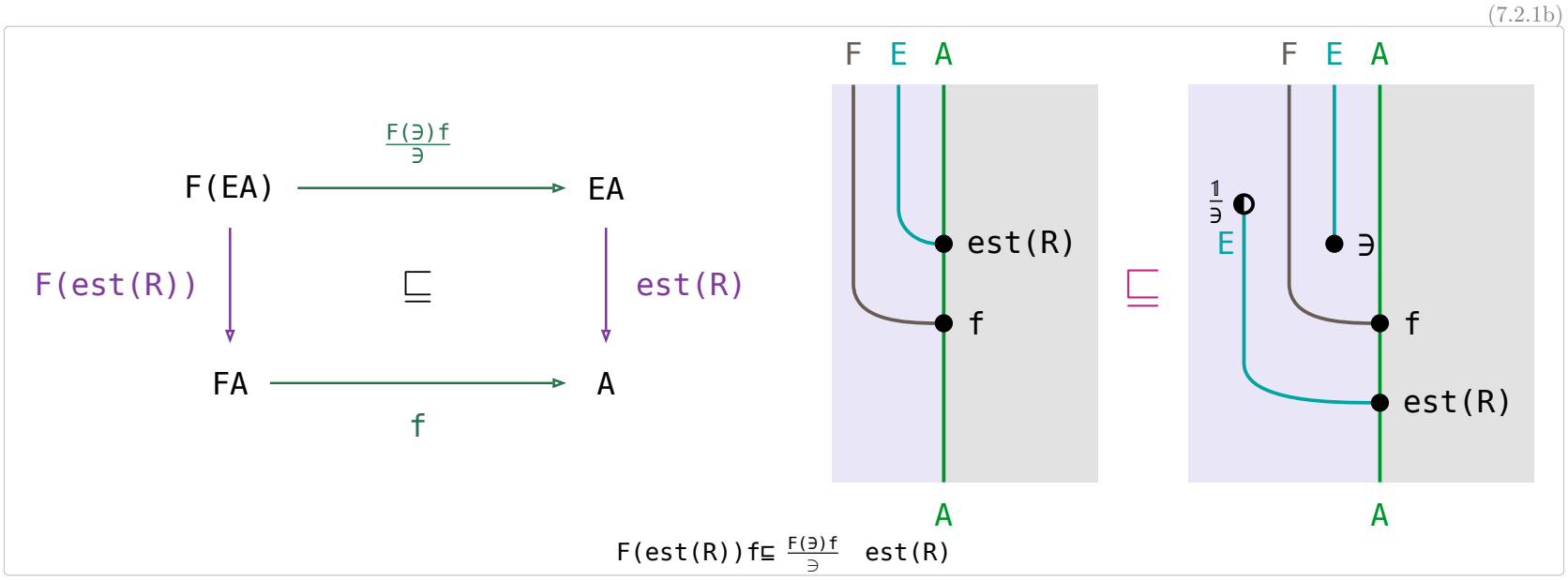
§7.2.1. Function f is monotonic on R iff it distributes over R

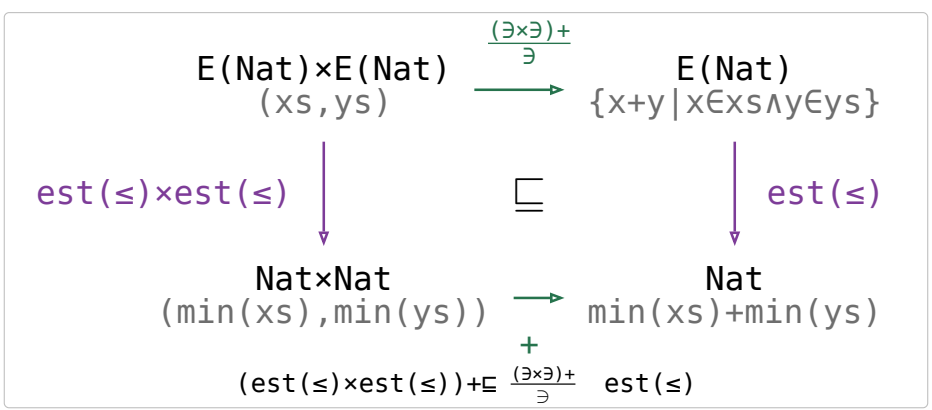
(7.2.1a)

$f : FA \to A$ **distributes over** R if $F(est(R))f \sqsubseteq \frac{F(\ni)f}{\ni} est(R)$.

$+$ distributes over $\leq$, at the point level $min(xs)+min(ys)$ is a least element of $\{x+y \mid x \in xs \wedge y \in ys\}$ for xs, ys non-empty and $min \triangleq est(\leq)$.

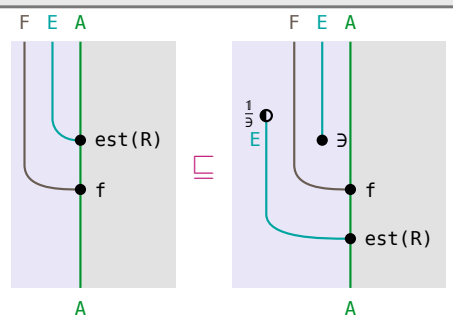
(7.2.1b)



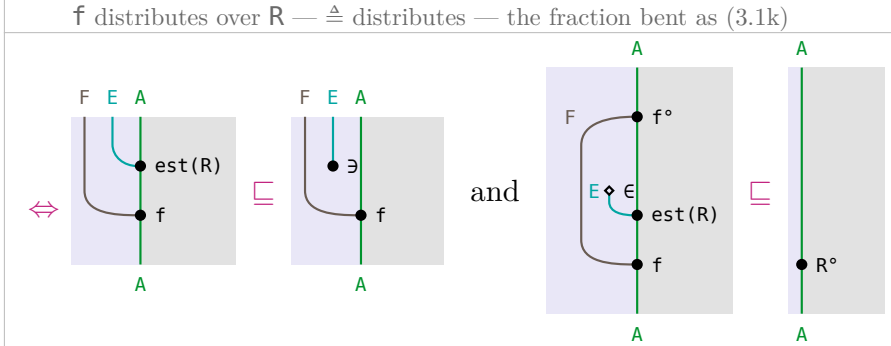


$$\mathsf{f}\circ\mathsf{F}(\mathsf{R})\mathsf{f}\sqsubseteq\mathsf{R}\Longleftrightarrow\mathsf{F}(\mathsf{est}(\mathsf{R}))\mathsf{f}\sqsubseteq\frac{\mathsf{F}(\ni)\mathsf{f}}{\ni}\ \mathsf{est}(\mathsf{R})$$ (7.2.1c)

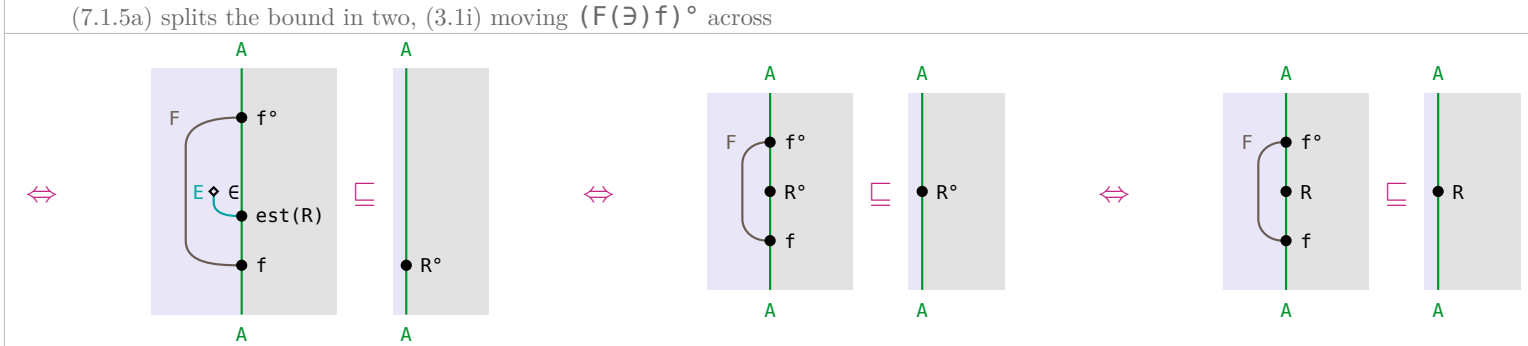
function f is monotonic over R if and only if it distributes over R; f a map, R reflexive



f distributes over R — ≜ distributes — the fraction bent as (3.1k)



(7.1.5a) splits the bound in two, (3.1i) moving $(\mathsf{F}(\ni)\mathsf{f})^\circ$ across



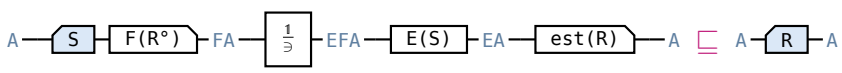
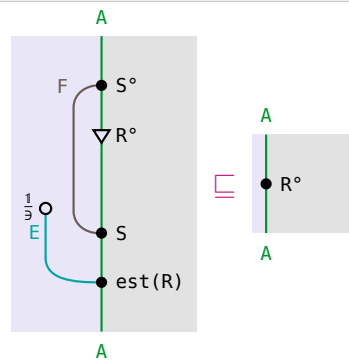
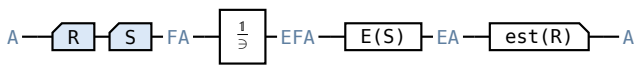
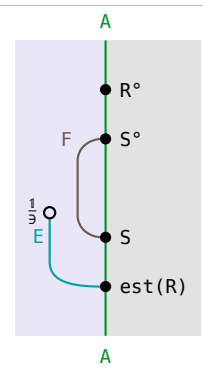
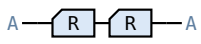
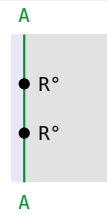
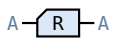
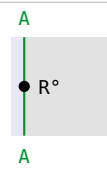
$\mathsf{est}(\mathsf{R})\sqsubseteq\ni$ — ≜ est — so the first conjunct drops

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

both sides conversed — $\mathsf{F}(\mathsf{R}^\circ)^\circ=\mathsf{F}(\mathsf{R})$, (5.1b)

§7.2.2. Greedy Theorem: $(\!|\frac{S}{\ni}\ \mathsf{est}(R)|\!) \sqsubseteq \frac{(\!|S|\!)}{\ni}\ \mathsf{est}(R)$, given S monotoic on R, F preserving °, and R transitive

(7.2.2a)

circuit	Hinze–Marsden
$(\!|\frac{S}{\ni}\ \mathsf{est}(R)|\!) \sqsubseteq \frac{(\!|S|\!)}{\ni}\ \mathsf{est}(R)$ the conclusion: one minimum kept at each step is below every result collected and one minimum taken at the end	
$\Leftrightarrow$ $(\!|\frac{S}{\ni}\ \mathsf{est}(R)|\!) \sqsubseteq (\!|S|\!)$ and $(\!|S|\!)^\circ(\!|\frac{S}{\ni}\ \mathsf{est}(R)|\!) \sqsubseteq R^\circ$ (7.1.5a) at $(\!|S|\!)$; $(\!|\frac{S}{\ni}\ \mathsf{est}(R)|\!) \sqsubseteq (\!|S|\!)^\circ \backslash R^\circ \Longleftrightarrow (\!|S|\!)^\circ(\!|\frac{S}{\ni}\ \mathsf{est}(R)|\!) \sqsubseteq R^\circ$ — (3.1i)	
$\Leftarrow$ $\frac{S}{\ni}\ \mathsf{est}(R) \sqsubseteq S$ the left conjunct; (7.1.5a), and $(\!|-|\!)$ is monotone	
 $\Leftarrow$ the right conjunct; $(\!|S|\!)^\circ(\!|\frac{S}{\ni}\ \mathsf{est}(R)|\!)$ is the least X with $X = S^\circ F(X)(\!|\frac{S}{\ni}\ \mathsf{est}(R)|\!)$ hylomorphism theorem — so Knaster–Tarski leaves this one inequation $\frac{S}{\ni} = \frac{1}{\ni}\ E(S)$ — (3.1k)	
$\sqsubseteq$  $S^\circ F(R^\circ) \sqsubseteq R^\circ S^\circ$ — $\triangleq$ monotonic at S, conversed; $F(R)^\circ = F(R^\circ)$ — (5.1b)	
$\sqsubseteq$  $S^\circ(\frac{S}{\ni}\ \mathsf{est}(R)) \sqsubseteq S^\circ(S^\circ \backslash R^\circ) \sqsubseteq R^\circ$ — (7.1.5a), (3.1i)	
$\sqsubseteq$  R transitive	

§7.2.3. takewhile(p)=⦇[nil,(π₁p→cons,⊸ nil)]⦈

(7.2.3a)

name	definition	type	example	in words
F	FX=1+A×X	$\mathcal{A}\to\mathcal{A}$		nothing, or a head and a tail
nil, cons	[A]::=nil|cons(A,[A]) ≜ subseq	1→[A] , A×[A]→[A]	cons(3, [1,2])=[3,1,2]	the empty list; a head onto a tail
α	[nil,cons]	F[A]→[A]		both constructors as one map
p	a coreflexive	A→A	p≜even — 2 p 2, and 3∉dom(p)	{(a,a)|a passes the test}
R	length≤length°, a preorder	[A]→[A]	[1] R [1,2]	xs R ys⟺length(xs)≤length(ys)
⊸ nil	the constant nil — the second nil of prefix	A×[A]→[A]	(⊸ nil)(3, [1,2])=nil	drop the pair, return nil
prefix	⦇[nil,⊸ nil ∪ cons]⦈ ≜ subseq	[A]→[A]	[3,1,2] prefix [3,1]	xs prefix ys⟺∃zs. xs=ys♯zs — at each cons, stop or keep the head
S	[nil,⊸ nil ∪ (p×𝟙) cons]	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
(g→X,Y)	X where g is defined and Y where it is not	A→B		the test picks the branch

nil R=⊤, nil R°=nil nil is the shortest list — below every list, and above only itself, so it loses every est(R°)

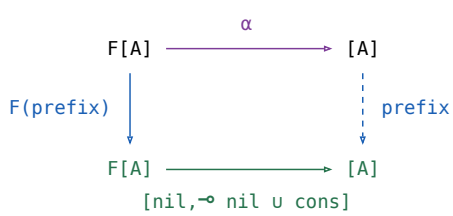
(7.2.3b)

prefix=⦇[nil,⊸ nil ∪ cons]⦈

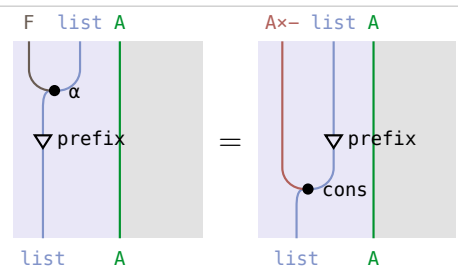
the fold whose algebra, at each **cons**, stops with **nil** or keeps the head: **xs prefix ys⟺∃zs. xs=ys♯zs**

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

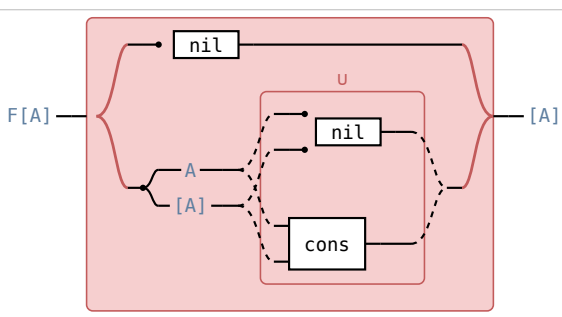
Hinze–Marsden



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



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



the algebra [nil,⊸ nil ∪ cons] : 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′)

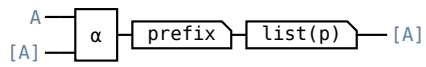
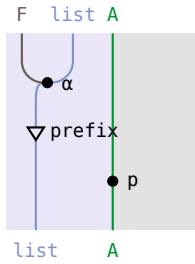
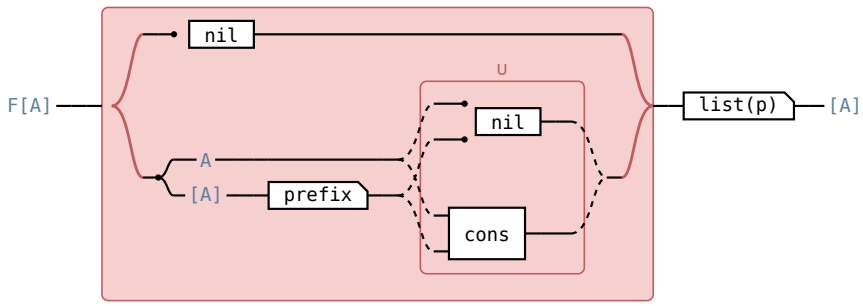
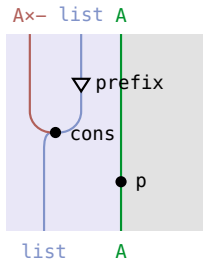
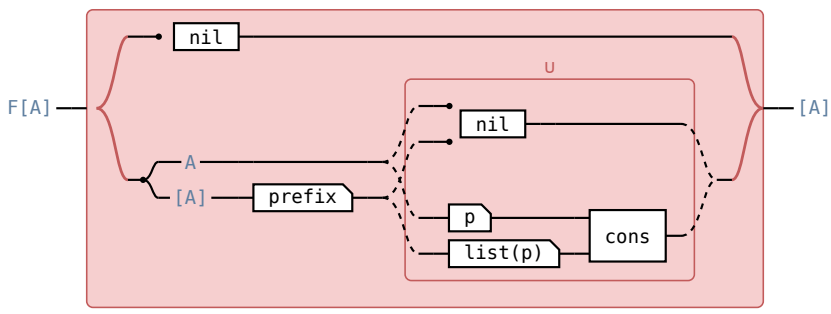
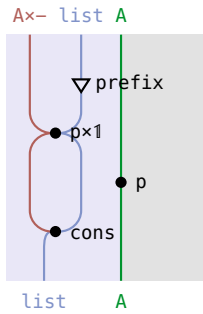
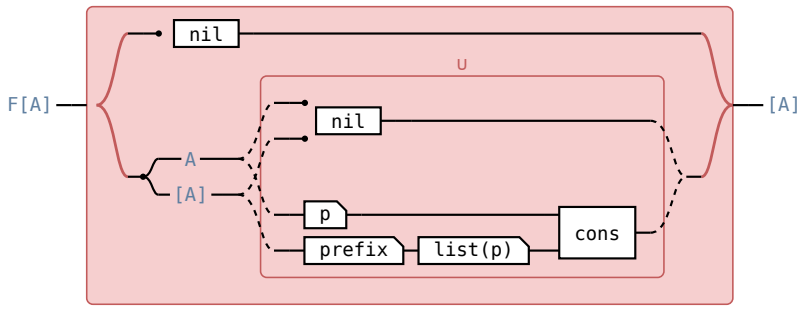
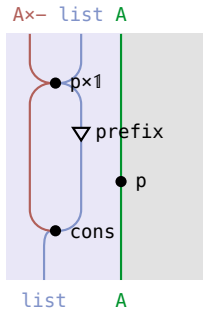
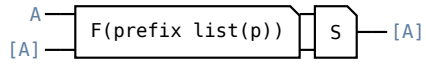
a right fold: α° peels, F(prefix) recurses, the algebra runs on the way back; init* (≜ subseq) is the tail-recursive form

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

αprefix list(p)=F(prefix list(p))S (7.2.3c)

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

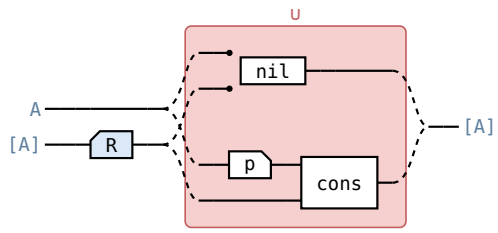
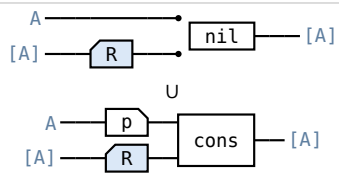
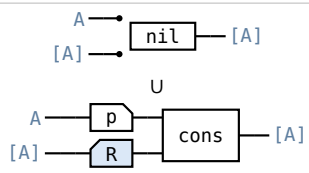
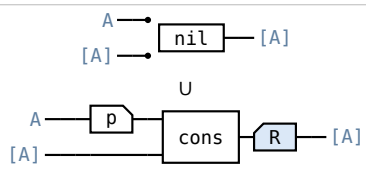
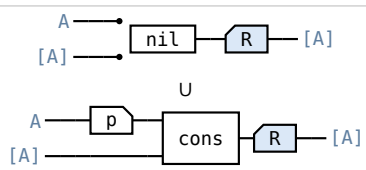
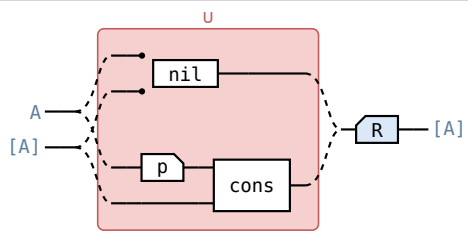
this same diagram is **subseq**'s: algebra **[nil,π₂ ∪ cons]**, type **[A]→[A]**

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,⊸ nil ∪ cons] list(p) defining equation	 the cons operand of ⊸ nil ∪ cons
=  F(prefix) [nil,⊸ nil ∪ (p×list(p)) cons] list(p) through cons	 the (p×list(p)) cons operand of ⊸ nil ∪ (p×list(p)) cons
=  [nil,⊸ nil ∪ (p×(prefix list(p))) cons] 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. ,

(𝟙×R°)(⊸ nil ∪ (p×𝟙)cons)⊑(⊸ nil ∪ (p×𝟙)cons)R° (7.2.3d)

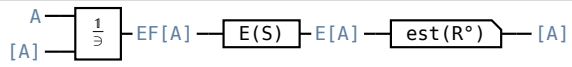
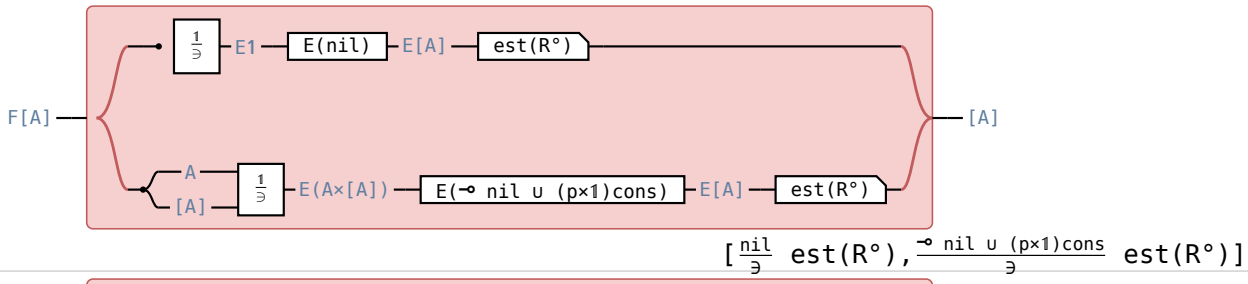
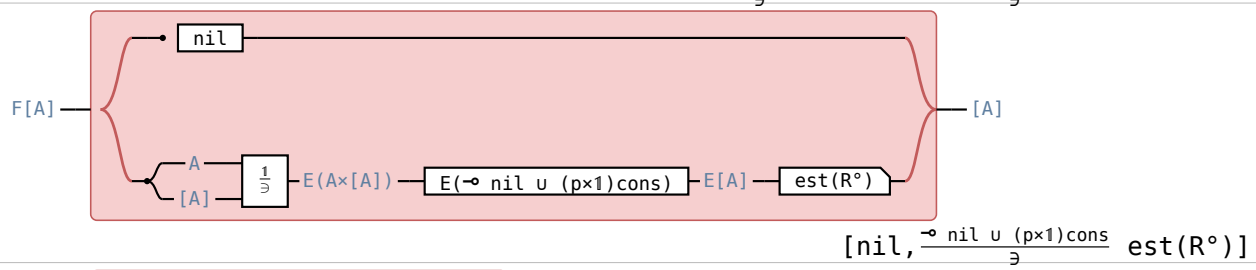
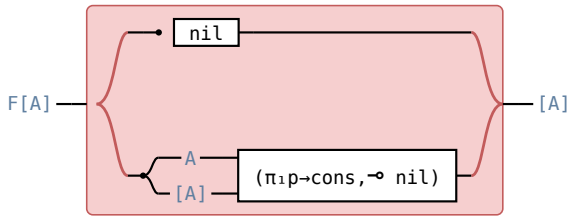
shortening the tail and then taking the step lands inside taking the step and then shortening the result

circuit — the cons branch of F(R°)S⊑SR°	**reason**
 (𝟙×R°)(⊸ nil ∪ (p×𝟙)cons)	
=  (𝟙×R°)⊸ nil ∪ (p×R°)cons	each operand is reached on its own (3.1a) — and (𝟙×R°)(p×𝟙) is p and R° on the pair's two strands at once
⊑  ⊸ nil ∪ (p×R°)cons	⊸ is the greatest arrow into 𝟙, so (𝟙×R°)⊸⊑⊸
⊑  ⊸ nil ∪ (p×𝟙)cons R°	cons length=(𝟙×length)π₂ succ with succ monotone — a shorter tail makes a shorter list
=  ⊸ nil R° ∪ (p×𝟙)cons R°	nil R°=nil ≜ takewhile — so the constant branch may carry the R° the other one already has
=  (⊸ nil ∪ (p×𝟙)cons)R°	one R° past the join is the two inside it (3.1a)

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

$\frac{S}{\ni}$ est(R°)=[nil,(π₁p→cons,⊸ nil)] (7.2.3e)

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=T

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.

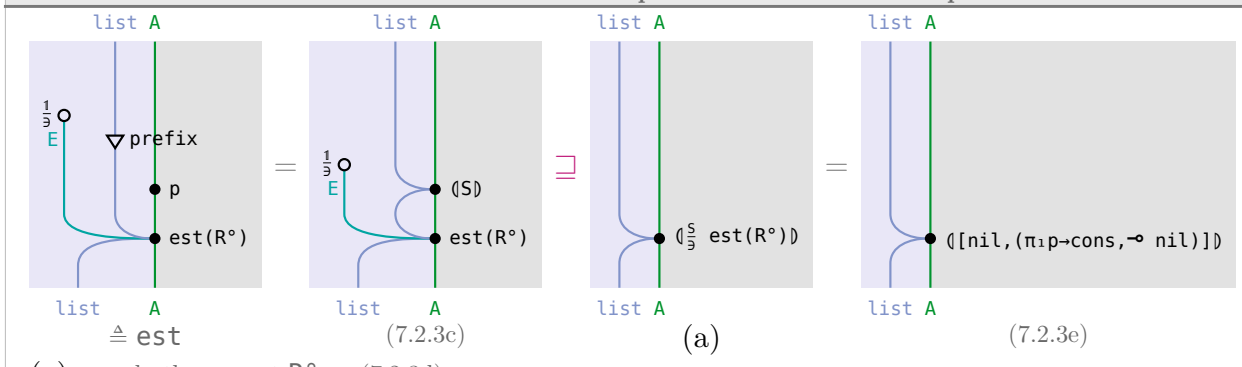
(7.2.3f)

name	definition	type	example	in words
S	[nil,⊸ nil ∪ (p×𝟙) cons] ≜ takewhile	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

takewhile(p)≜$\frac{\text{prefix list(p)}}{\ni}$ est(R°) (7.2.3g)

takewhile(p)=⦇[nil,(π₁p→cons,⊸ nil)]⦈

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.



(a) greedy theorem at R° — (7.2.3d)

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

takewhile(p)⊑prefix list(p) and (prefix list(p))° takewhile(p)⊑R — (7.1.5a) at est(R°) — and two prefixes of one list of equal length are equal, so takewhile(p) is simple: **the** longest, not **a** longest.

$\frac{\text{prefix list(p)}}{\ni}$ 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 =.

§7.2.4. mss=⦇[zero⟨𝟙,$\frac{1}{\cup}$⟩,⟨(𝟙×π₁)⊕,⟨(𝟙×π₁)⊕ $\frac{1}{\cup}$,π₂π₂⟩ cup⟩]⦈ π₂ est(≥)

(7.2.4a)

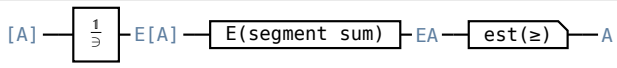
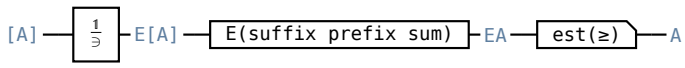
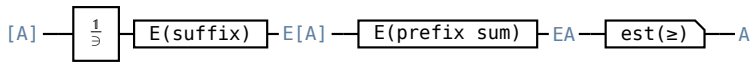
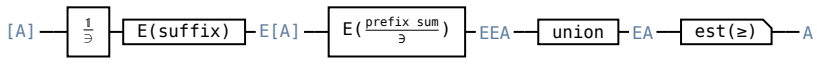
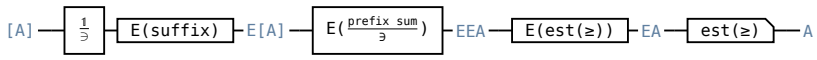
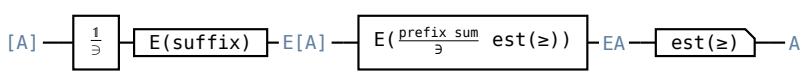
FX=1+A×X, α≜[nil,cons], sum=⦇[zero,plus]⦈ and segment=suffix prefix from (3.1m) and ≜ subseq.

head≜cons° π₁, wrap≜⟨𝟙,⊸ nil⟩ cons the head of a list and the one-element list, beside ≜ subseq's tail≜cons° π₂

⊕≜ ⊸ zero ∪ plus / ∋ est(≥) the set at (a,b) is {0,a+b}, so ⊕ is the larger of the two,

mss=suffix/∋ E(prefix sum/∋ est(≥))est(≥) (7.2.4b)

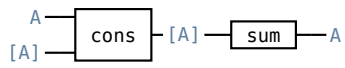
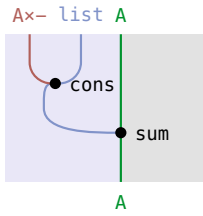
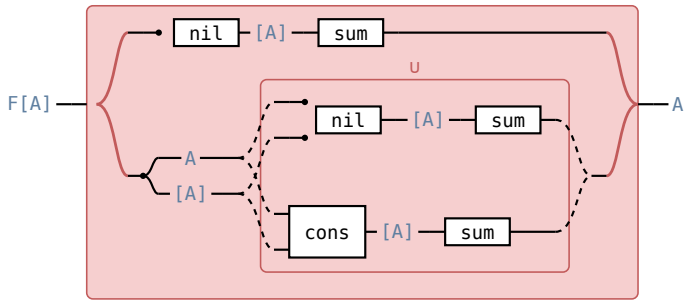
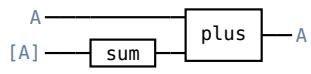
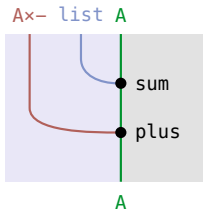
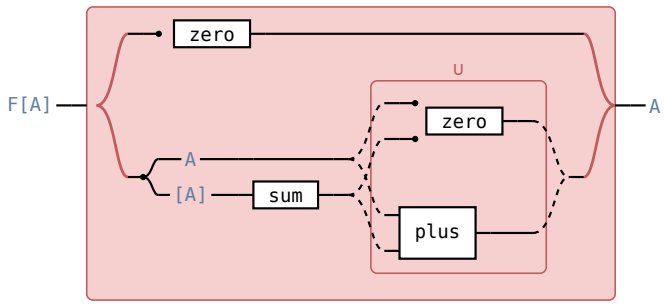
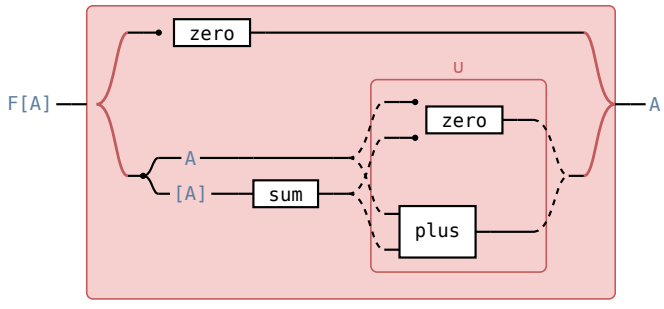
maximum segment sum problem: using **segment=suffix prefix**, the specification is expressed in this form

formula — one wire from [A] to A, its type written along it	**reason**
	
= 	segment=suffix prefix ≜ subseq, ≜ mss
= 	absorption (3.1j) — frac(S,∋) E(R)=frac(SR,∋) at S:=suffix, R:=prefix sum
= 	R/∋ ∋=R, union=E(∋) (3.1j)'s frac(R,∋)∋=R at R:=prefix sum and E(R)≜frac(∋R,∋); (7.1b)'s union≜frac(∋∋,∋); the middle equality is ≜ relator's F(RS)=F(R)F(S) at F:=E
= 	(7.1b), the sets non-empty
= 	relator ≜ relator — F(RS)=F(R)F(S) at F:=E

The union step is (7.1b)'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 ((5.4d)).

[nil,⊸ nil ∪ cons]sum=F(sum)S (7.2.4c)

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

circuit — the fork is the bracket's case split F([A])=𝟙+A×[A]: nil above, the pair and its ∪ below	**Hinze–Marsden**
 [nil,⊸ nil ∪ cons] sum	 the cons operand of ⊸ nil ∪ cons
=  [nil sum,⊸ nil sum ∪ cons sum] coproduct of maps, composition over ∪	
=  [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	

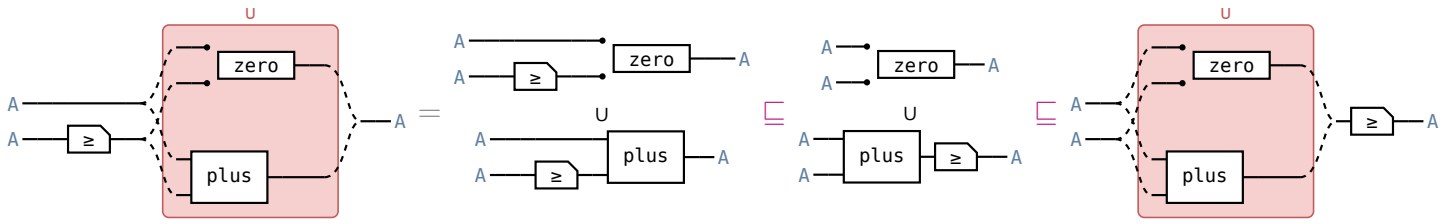
fusion at R:=[nil,⊸ nil ∪ cons], S:=sum: the side condition, so prefix sum=⦇[zero,⊸ zero ∪ plus]⦈. prefix is the reduce, sum the map fused into it — the intermediate list is gone.

(𝟙×≥)(⊸ zero ∪ plus)⊑(⊸ zero ∪ plus)≥ (7.2.4d)

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 ⊑ ⊸ zero ∪ plus≥ ⊑ (⊸ zero ∪ plus)≥



relator, composition over ∪

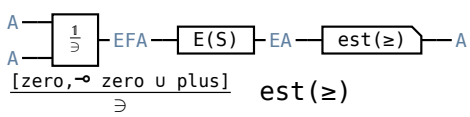
(3.1h), (≥×≥) plus⊑plus≥; (≤×≤) plus⊑plus≤ is ≜ monotonic, written + there, and plus is a map, so it is monotonic on an order and on its opposite together, which carries it to ≥.

≥ reflexive

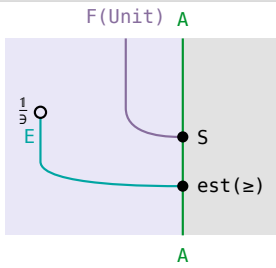
⅀ est(≥)=[zero,⊕] (7.2.4e)

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

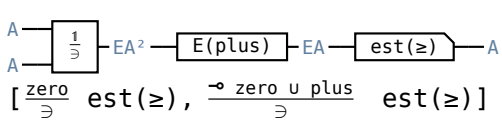
circuit — the tape is the coproduct: zero's branch above, plus's below



Hinze–Marsden

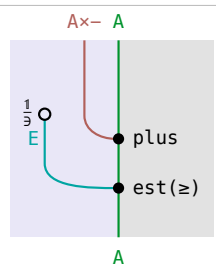


the zero arm of the bracket

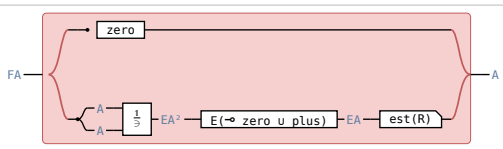


= [zero/∋ est(≥), ⊸ zero ∪ plus/∋ est(≥)]

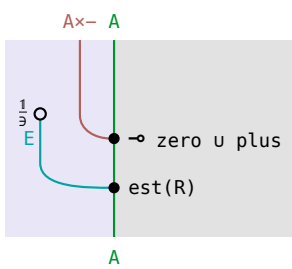
coproduct of maps — (5.3.1a) at T:=[zero,⊸ zero ∪ plus], then [U,V]Z=[UZ,VZ] — (5.3a), composition over ∪



the plus operand of the lower arm's ⊸ zero ∪ plus, under its 𝟙%∋ E(…) and est(≥)



= singleton, ≥ reflexive — (7.1b)'s 𝟙/∋ est(R)=𝟙∩R at R:=≥, zero a map; the lower branch is ⊕'s definition, ≜ mss, and no law



with (7.2.4d) the greedy theorem gives ⦇[zero,⊕]⦈⊑ prefix sum/∋ est(≥) — and (7.2.4g) makes it an equality.

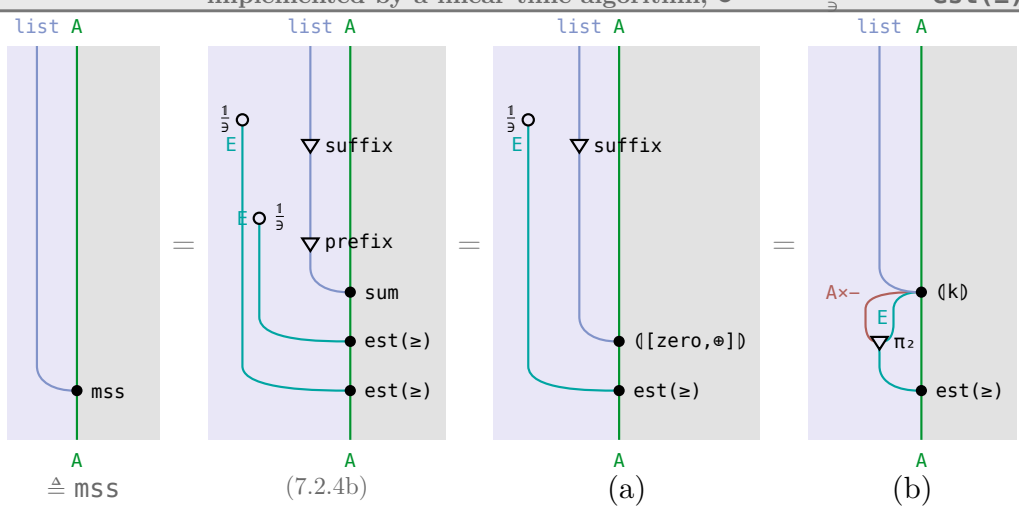
$$[\mathsf{nil},\mathsf{cons}]\langle(\!|[\mathsf{zero},\oplus]|\!),\overset{\mathsf{suffix}}{\ni}\ \mathsf{E}(\!|[\mathsf{zero},\oplus]|\!)\rangle=\mathsf{F}(\langle(\!|[\mathsf{zero},\oplus]|\!),\overset{\mathsf{suffix}}{\ni}\ \mathsf{E}(\!|[\mathsf{zero},\oplus]|\!)\rangle)\mathsf{k} \tag{7.2.4f}$$

$\mathsf{k}\triangleq[\mathsf{zero}\langle\mathbb{1},\overset{1}{\ni}\rangle,\langle\mathsf{w},\langle\mathsf{w}\ \overset{1}{\ni},\pi_2\pi_2\rangle\ \mathsf{cup}\rangle]$, $\mathsf{w}\triangleq(\mathbb{1}\times\pi_1)\oplus$: 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
$\mathsf{nil}\langle\mathsf{g},\overset{\mathsf{suffix}}{\ni}\ \mathsf{E}(\mathsf{g})\rangle=\mathsf{zero}\langle\mathbb{1},\overset{1}{\ni}\rangle$	nil has one suffix, itself, and $\mathsf{g}(\mathsf{nil})=\mathsf{zero}$, so the set is the singleton $\{\mathsf{zero}\}$
$\mathsf{cons}\langle\mathsf{g},\overset{\mathsf{suffix}}{\ni}\ \mathsf{E}(\mathsf{g})\rangle=(\mathbb{1}\times\langle\mathsf{g},\overset{\mathsf{suffix}}{\ni}\ \mathsf{E}(\mathsf{g})\rangle)\langle\mathsf{w},\langle\mathsf{w}\ \overset{1}{\ni},\pi_2\pi_2\rangle\ \mathsf{cup}\rangle$	$\mathsf{g}(\mathsf{cons}(\mathsf{a},\mathsf{x}))=\mathsf{a}\oplus(\mathsf{g}(\mathsf{x}))$, which is w reading $\mathsf{g}(\mathsf{x})$ off π_1; and the suffixes of $\mathsf{cons}(\mathsf{a},\mathsf{x})$ are $\mathsf{cons}(\mathsf{a},\mathsf{x})$ 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

$$\mathsf{mss}=(\!|\mathsf{k}|\!)\pi_2\ \mathsf{est}(\geq) \tag{7.2.4g}$$

maximum segment sum problem: $\overset{\mathsf{suffix}}{\ni}\ \mathsf{E}(\!|[\mathsf{zero},\oplus]|\!)$ expressed as the catamorphism $(\!|\mathsf{k}|\!)$, hence mss implemented by a linear-time algorithm, $\oplus\triangleq\overset{\mathsf{zero}\ \cup\ \mathsf{plus}}{\multimap_{\ni}}\ \mathsf{est}(\geq)$ — $\triangleq\mathsf{mss}$; k and w — (7.2.4f).



(a) greedy theorem at $\mathsf{R}:=\geq$ — (7.2.4d), (7.2.4e), (7.2.3g)

(b) $(\!|\mathsf{k}|\!)=\langle(\!|[\mathsf{zero},\oplus]|\!),\overset{\mathsf{suffix}}{\ni}\ \mathsf{E}(\!|[\mathsf{zero},\oplus]|\!)\rangle$ — $\triangleq\mathsf{reduce}$, (7.2.4f)

one fold builds the $\mathsf{n+1}$ running maxima and the final $\mathsf{est}(\geq)$ reads them in one more pass, so mss is linear.

§7.2.5. filter(p)=⦇[nil,(π₁p→cons,π₂)]⦈

(7.2.5a)

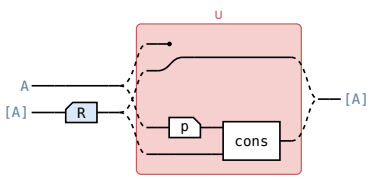
name	definition	type	example	in words
F, α, p, R	as in ≜ takewhile			
π₂	π₂ where ≜ takewhile has ⊸ nil	A×[A]→[A]	π₂(3,[1,2])=[1,2]	drop the head, keep the tail
subseq	⦇[nil,π₂ ∪ cons]⦈ ≜ subseq	[A]→[A]	[3,1,2] subseq [3,2]	xs subseq ys⟺ys is xs with elements dropped — at each cons, drop the head or keep it
S	[nil,π₂ ∪ (p×𝟙) cons]	F[A]→[A]	(4,[2]) S [2] and (4,[2]) S [4,2], but (3,[2]) S [2] only	subseq's algebra with one extra p — drop the head, or keep a head that passes p
𝟙⊑π₂R cons°				the tail is one shorter than the cons, so π₂ loses the est(R°) at every step — where ≜ takewhile's loser is nil

(𝟙×R°)(π₂ ∪ (p×𝟙) cons)⊑(π₂ ∪ (p×𝟙) cons)R° (7.2.5b)

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

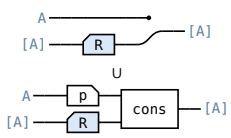
formula — the cons branch of F(R°)S⊑SR°; **reason** under each circuit

(𝟙×R°)(π₂ ∪ (p×𝟙)cons)



=

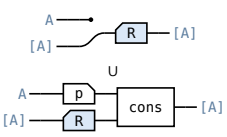
(𝟙×R°)π₂ ∪ (p×R°)cons



(𝟙×R°)(p×𝟙)=p×R° (3.1a)

=

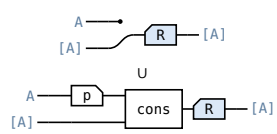
π₂ R° ∪ (p×R°)cons



(𝟙×R°)π₂=π₂R° (5.8.1c)

⊑

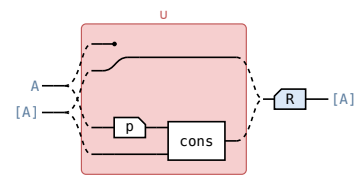
π₂ R° ∪ (p×𝟙)cons R°



(p×R°) cons⊑(p×𝟙) cons R° (7.2.3d)

⊑

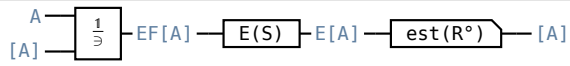
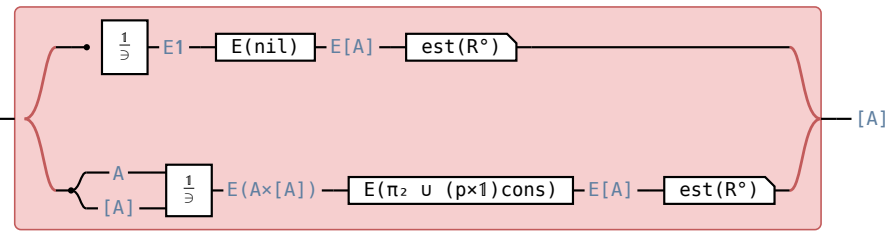
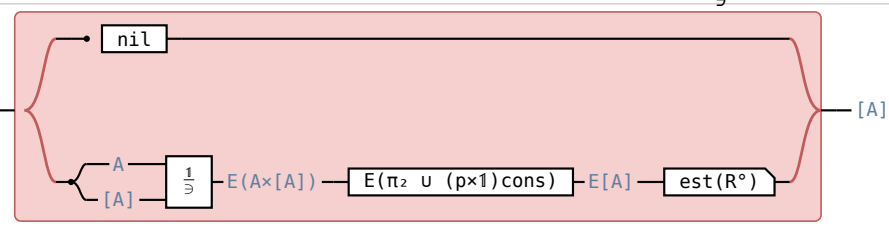
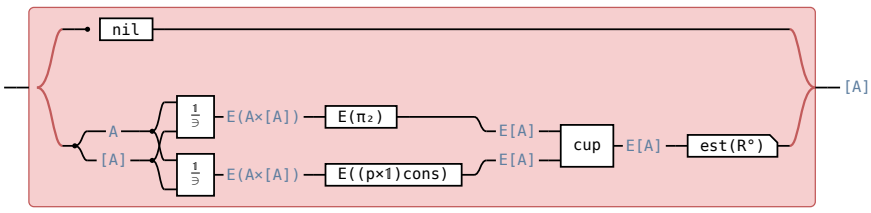
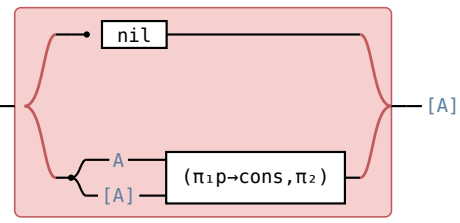
(π₂ ∪ (p×𝟙)cons)R° (3.1a)



the nil branch: nil⊑nil R°.

$\frac{S}{\ni}$ est(R°)=[nil,(π₁p→cons,π₂)] (7.2.5c)

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
	
=  $[\frac{nil}{\ni}\ est(R°), \frac{\pi_2 \cup (p\times\mathbb{1})cons}{\ni}\ est(R°)]$	S=[nil,π₂ ∪ (p×𝟙) cons] ≜ filter — and the %∋ of a coproduct of maps is the coproduct of their %∋ (5.3.1a)
=  $[nil, \frac{\pi_2 \cup (p\times\mathbb{1})cons}{\ni}\ est(R°)]$	nil%∋ is the singleton {nil}, and est(R°) of a singleton is its element because R° is reflexive ≜ est
= 	$\frac{\pi_2 \cup (p\times\mathbb{1})\ cons}{\ni} = \langle \frac{\pi_2}{\ni}, \frac{(p\times\mathbb{1})\ cons}{\ni} \rangle$ cup ≜ cup
= 	𝟙⊑π₂R cons° ≜ filter — 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

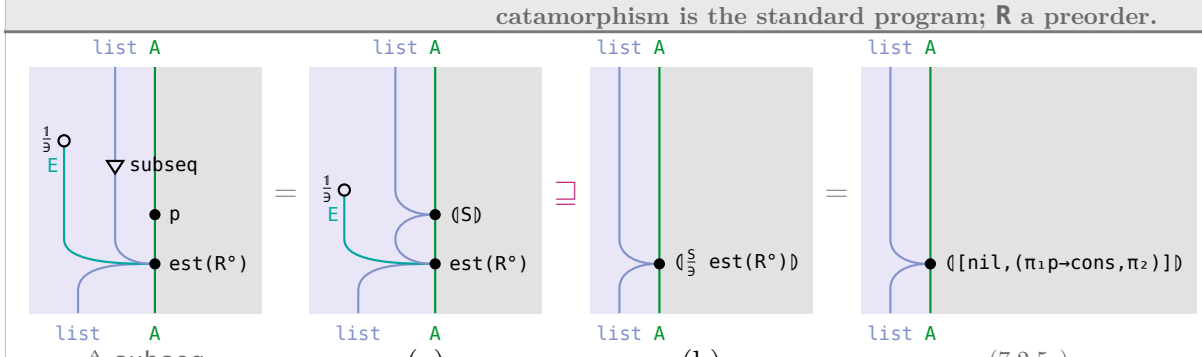
the head is dropped, not the whole tail: that is the one place π₂ shows against (7.2.3e)'s ⊸ nil.

(7.2.5d)

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

filter(p)=⦇[nil,(π₁p→cons,π₂)]⦈ (7.2.5e)

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.



(a) subseq list(p)=⦇S⦈ — (7.2.3c)
(b) greedy theorem at R° — (7.2.5b)

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 §7.2.3'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.

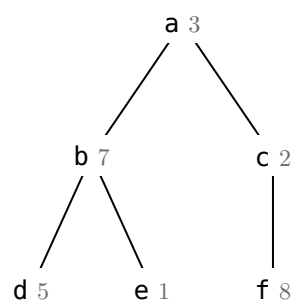
§7.3. Planning a company party

(7.3a)

definition	type	note
`tree A::=node (A,[tree A])`	$\mathcal{A}\to\mathcal{A}$	The company hierarchy: an employee, and the list of subtrees under them.
`F(A,B)=A×[B]`	$\mathcal{A}\to\mathcal{A}$	The base functor `tree` folds: an employee beside the recursive position, one layer deep.
`rating`	`A→Int`	What one employee is worth as a guest.
`cost=list(rating)sum`	`[A]→Int`	What a guest list is worth.
`R=cost≤cost°`	`[A]→[A]`	The preorder the guest list is maximised over.
`choose≜π₁ ∪ π₂`	`[A]×[A]→[A]`	Takes one of the two parties a subtree returns.
`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.
`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.
`S≜⟨include,exclude⟩`	`F(A,[A]×[A])→[A]×[A]`	The algebra: one step returns both parties of a subtree at once.
`party=⦇S⦈choose`	`tree(A)→[A]`	Every guest list the president's ruling allows.
the specification $\frac{\texttt{party}}{\ni}$ `est(R°)`	`tree(A)→[A]`	A guest list of greatest total conviviality.

§7.3.1. `include` and `exclude`

(7.3.1a)



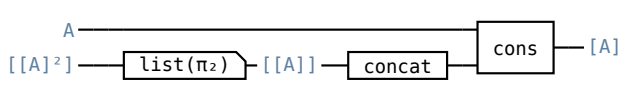
the small grey number is the employee's rating

(7.3.1b)

node	the algebra's input `A×[[A]×[A]]`	`include`	`exclude`
`d, e, f`	`(d,[])`	`[d]`	`[]`
`b`	`(b,[([d],[]),([e],[])])`	`[b]`	`[d,e], [d], [e], []`
`c`	`(c,[([f],[])])`	`[c]`	`[f], []`
`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
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`
2nd, `([c],[f])`	`[c] R° [c]` 2≥2, reflexivity	`[f] R° []` 8≥0, and `([c],[f])` is the pair `est(R°)` keeps at `c`

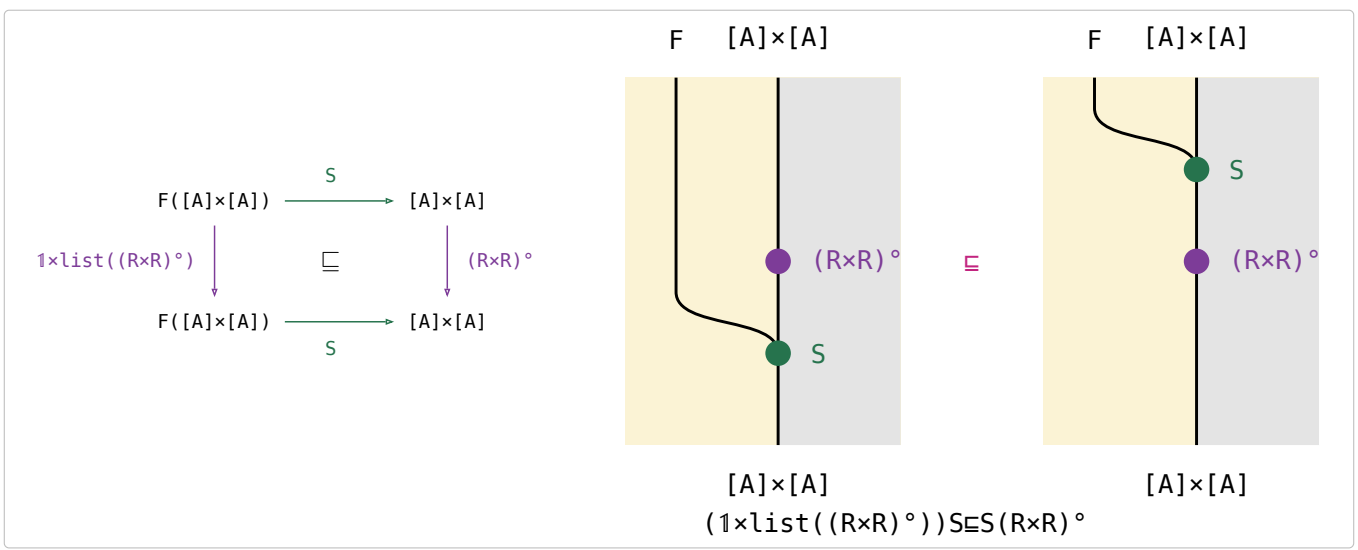
the first component of `𝟙×list((R×R)°)` is the root employee.

`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.

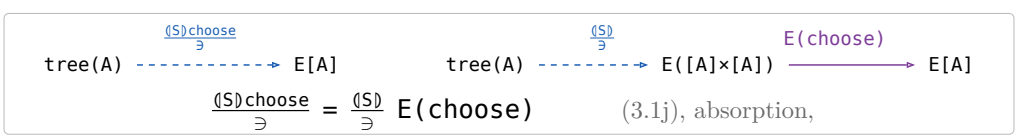
`R` compares cost only, not contents: `[f] R° []` is legal though `[f]` has an element `[]` does not. That is why the three leaves of §7.3.3's proof all reduce to "`cost` is a sum".

§7.3.3. `(𝟙×list((R×R)°))S⊑S(R×R)°` — `S : F([A]×[A])→[A]×[A]` monotonic on `(R×R)°`

(7.3.3a)

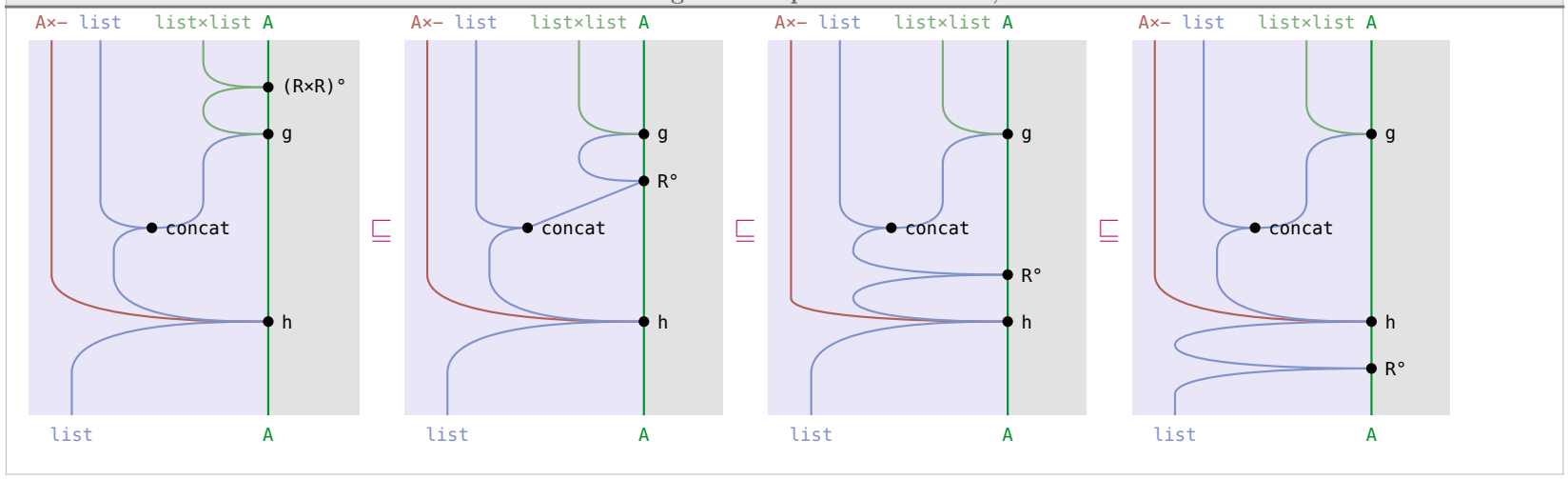


(7.3.3b)



$$S \triangleq (\text{include}, \text{exclude})$$
$$(\mathbb{1}\times\text{list}((R\times R)^\circ))S \sqsubseteq S(R\times R)^\circ$$
$$(\mathbb{1}\times\text{list}((R\times R)^\circ))\text{include} \sqsubseteq \text{include } R^\circ$$
$$(\mathbb{1}\times\text{list}((R\times R)^\circ))\text{exclude} \sqsubseteq \text{exclude } R^\circ \qquad (7.3.3c)$$

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,



g	$(R\times R)^\circ g \sqsubseteq g R^\circ$ $g := \pi_2$ 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 $(\text{dom}(\pi_2 R^\circ))\pi_1 R^\circ \sqsubseteq \pi_1 R^\circ$ — 1 and 4 of (5.2.2b), then $\text{dom} \sqsubseteq \mathbb{1}$; $g := \text{choose} \triangleq \pi_1 \cup \pi_2$ is the union of the two (5.9c). list monotonic, $\triangleq$ relator
concat	$\text{list}(R^\circ)\text{concat} \sqsubseteq \text{concat } R^\circ$ 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	$(\mathbb{1}\times R^\circ)h \sqsubseteq h R^\circ$ 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 $\text{dom}(\pi_1) = \mathbb{1}$ — (3.1g)

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

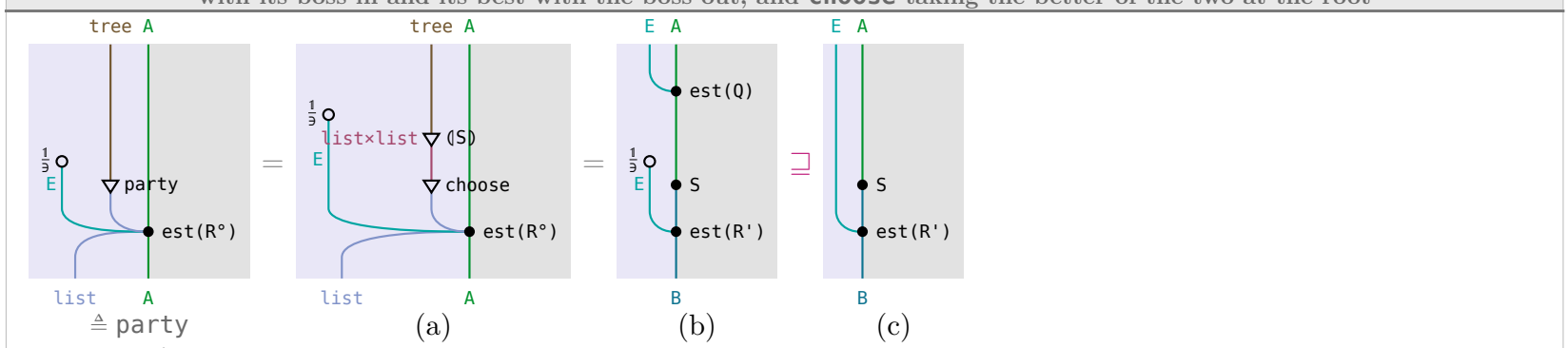
(7.3.3d)

$R \triangleq \{(0,0)\} : \{0,1\} \to \{0,1\}$	not entire — undefined at 1
$(R\times R)\text{choose}$ at $(0,1)$	nothing — $R\times R$ needs BOTH components related, and R is undefined at 1
choose R at $(0,1)$	0 — choose picks the first component 0, and R 0 0 holds
$((0,1),0)$	in choose R, not in $(R\times R)\text{choose}$
$(R\times R)\text{choose} = \text{choose } R$	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. The derivation

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

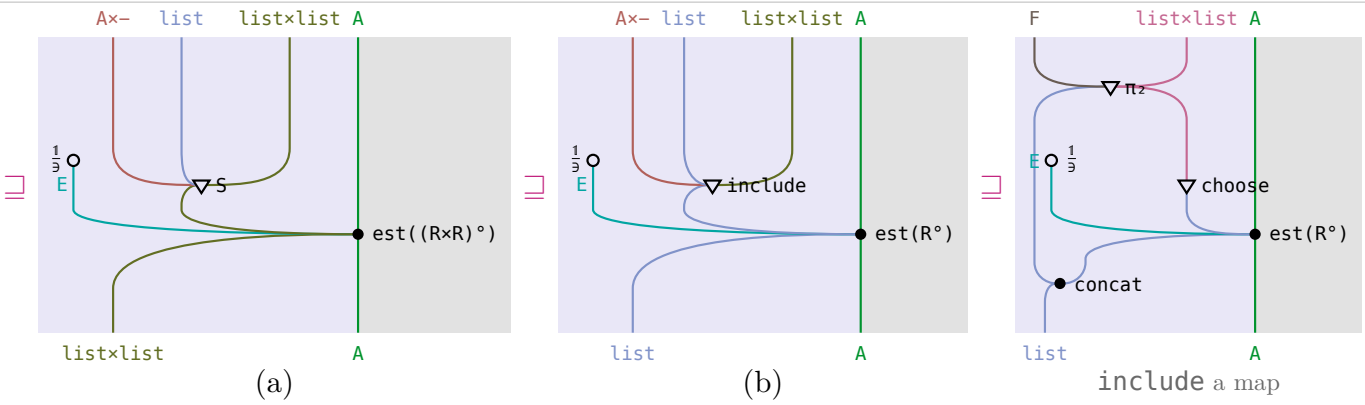
the best of every guest list the president allows is one pass up the tree, each subtree handing up its best party with its boss in and its best with the boss out, and **choose** taking the better of the two at the root



$\triangleq$ party (a) (b) (c)

(a) $\text{party} \triangleq (\!|S|\!)\ \text{choose}$ — $\triangleq$ party
(b) $\frac{(\!|S|\!)\ \text{choose}}{\ni} = \frac{(\!|S|\!)}{\ni}\ E(\text{choose})$ — (7.3.3b)
(c) $(R\times R)^\circ \text{choose} \sqsubseteq \text{choose } R^\circ$ — (7.3.3c)

(7.3.4b)



(a) (b) `include` a map

(a) `(𝟙×list((R×R)°))S⊑S(R×R)°` — (7.3.3a)

(b) $\langle \frac{\texttt{include}}{\ni}\ \texttt{est(R°)}, \frac{\texttt{exclude}}{\ni}\ \texttt{est(R°)}\rangle \sqsubseteq \frac{\texttt{S}}{\ni}\ \texttt{est((R×R)°)}$

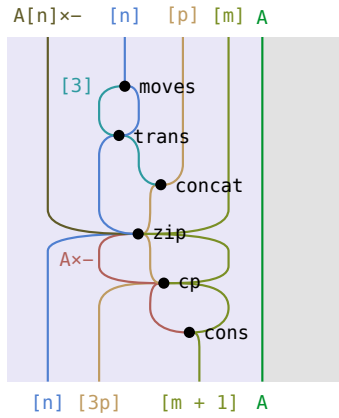
§7.4. Shortest paths on a cylinder

(7.4a)

	on L=list⁺, 11.3.3	on Vec	note
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.
the input	L(N A)	A[m][n]≜(A[n])[m]	m columns of n squares, outermost index first — the matrix itself.
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].
the paths into a row	E(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.
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.
moves	NA→E(NA)	X[n]→X[3][n]	moves(x)=(up(x),x,down(x)), a 3×n matrix.
trans	E(NX)→N(EX)	X[3][n]→X[n][3]	The transpose.
union	E(EX)→EX	concat:X[j][k]→X[jk]	The j rows laid end to end; j is 3 inside gen and n inside paths.
zip	F(NA,NB)→NF(A,B)	F(A[n],B[n])→F(A,B)[n]	Unchanged.
cp	F(A,EB)→E(F(A,B))	A×B[p]→(A×B)[p]	The square paired with each of the p candidates.
est(R)≜∋∩(∈\R℘)X→X		X[k]→X, ∋:X[k]→X	A component that is R-below every component; ∋ is "is some component" and ∈ its converse.
R≜sum≤sum°	L Nat→L Nat	Nat[m]→Nat[m]	The cost of a path, which the cheapest minimises — one order per length m.
gen	F(𝟙,moves trans N(union)) zip N(cp P(α)) F(NA,N(E(LA))) →N(E(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].
⦇gen⦈	L(N A)→N(E(L A))	A[m][n]→A[n][p][m]	Columns in, rows out.
paths≜⦇gen⦈ concat	⦇gen⦈ setify union L N Nat→E(L Nat)	A[m][n]→A[np][m]	The n rows of p paths laid end to end: every path of the cylinder, in one row.
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.
the specification paths est(R)	L N Nat→L Nat	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(n)(concat))) zip Vec(n)(cp Vec(3p)(cons))

(7.4.1a)

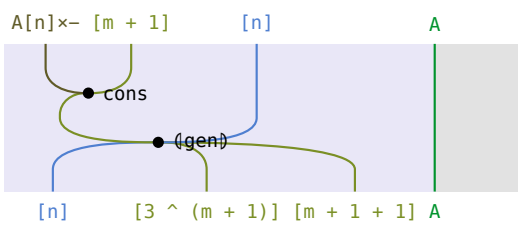


(7.4.1b)

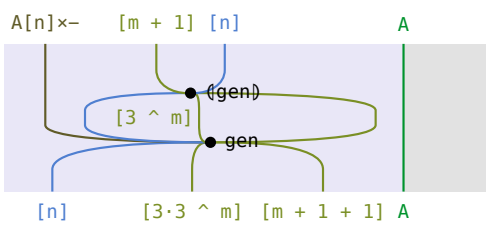
```
a row is the outer index, a cell one path
u = ((1,2,3,4),C)                                  : A[n]×A[n]
    C = 5                                            one candidate per row, one square long:
        6                                            a column of squares
        7
        8
(𝟙×(moves 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. gen is an F-algebra; ⦇gen⦈: cons ⦇gen⦈=(𝟙×⦇gen⦈)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)

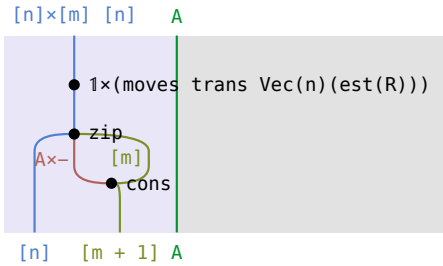
```
xs = 1 2 3 4                                       : A[m][n], m=2
     5 6 7 8

⦇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 (7.4.1b): 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, A[n]×A[n][m]→A[n][m+1]

(7.4.3a)



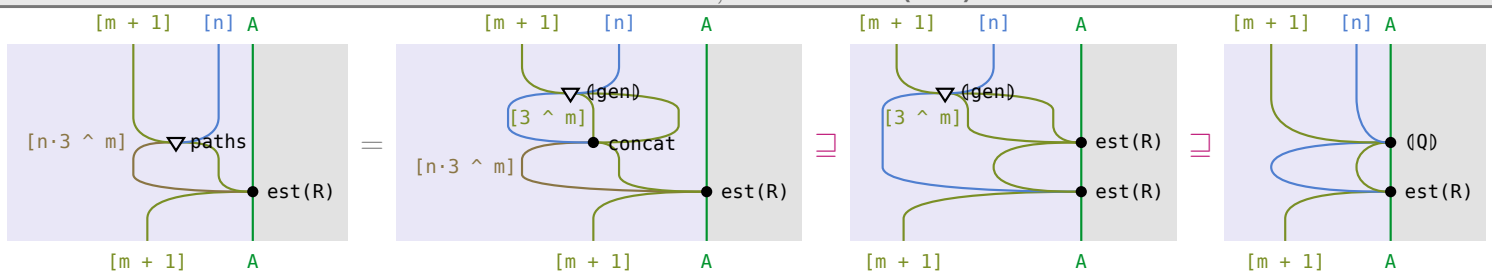
(7.4.3b)

```
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. paths est(R)⊒⦇Q⦈ est(R)

(∀i. 𝟙⊑R) ∧ (∀i. RR⊑R) ∧ (∀i. (𝟙×R)cons⊑cons R) ⟹ ⦇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)**.



(7.4a) at paths R transitive fusion at (7.4.4b)

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

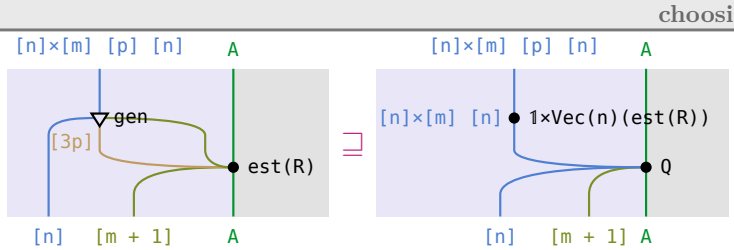
a cheapest of each of the **j** rows and then a cheapest of those **j** is a cheapest of all **jk** entries laid end to end.

(∀i. 𝟙⊑R) ∧ (∀i. RR⊑R) ∧ (∀i. (𝟙×R)cons⊑cons R) ⟹ ⦇Q⦈⊑⦇gen⦈Vec(n)(est(R)) , R reflexive, transitive and monotonic

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.

RR⊑R ∧ (𝟙×R)cons⊑cons R ⟹ (𝟙×Vec(n)(est(R)))Q⊑gen Vec(n)(est(R)) (7.4.4b)

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**.



(a)

(a) (7.13); zip, trans, moves lax natural

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

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.

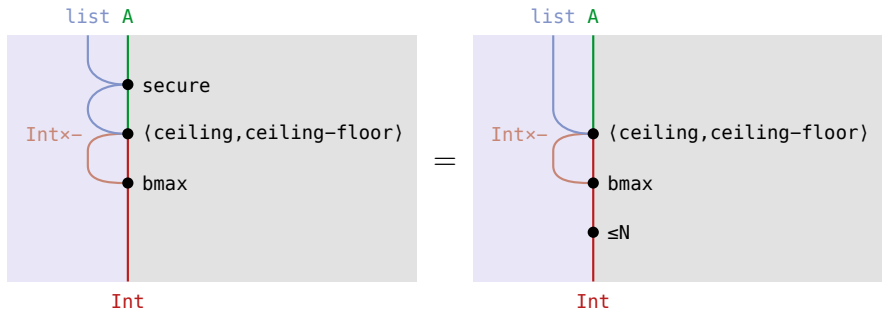
§7.5. The security van problem

(7.5a)

definition	type	example
R≜length≤length°	[[A]]→[[A]]	[[a,b,c]] to [[a],[b,c]], and not back: 1≤2.
ceiling≜ $\frac{\text{prefix sum}}{\ni}$ est(≥)	[A]→Int	ceiling[a,b] is the largest of 0, a, a+b.
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.
secure the coreflexive on x with bmax(ceiling x,ceiling x−floor x)≤N	[A]→[A]	secure keeps [a] exactly when −N≤a≤N.
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.
new≜(wrap×𝟙) cons	A×[[A]]→[[A]]	new(a,[[b],[c]])=[[a],[b],[c]].
glue≜(𝟙×cons°) assocl (cons×𝟙) cons	A×[[A]]→[[A]]	glue(a,[[b],[c]])=[[a,b],[c]].
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.
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]].
S≜[nil,new ∪ old]	F[[A]]→[[A]]	S(a,[[b],[c]]) gives [[a],[b],[c]], and [[a,b],[c]] when [a,b] is secure.
(∀a. secureP(amount,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.
H≜(head prefix° head°) ∪ (nil° nil)	[[A]]→[[A]]	[[a],[b,c]] to [[a,b],[c]]: [a] is a prefix of [a,b].
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.
|R|≜R∩¬R°	[[A]]→[[A]]	[[a,b,c]] to [[a],[b,c]]: 1<2. The strict part R splits into: R;H=|R| ∪ (R∩H).
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

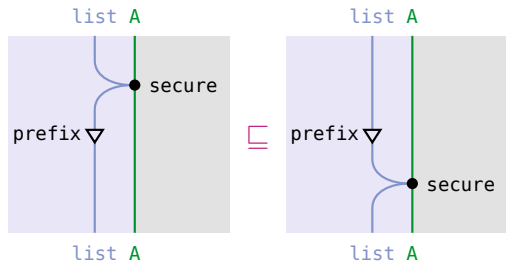
(7.5.1a)



`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`'s `bmax(ceiling x,ceiling x−floor x)≤N`

(7.5.1b)

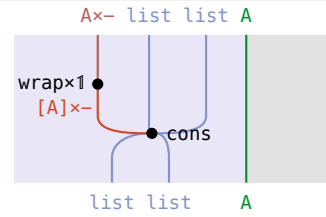
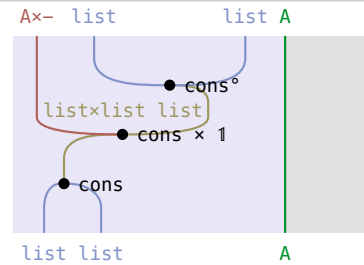
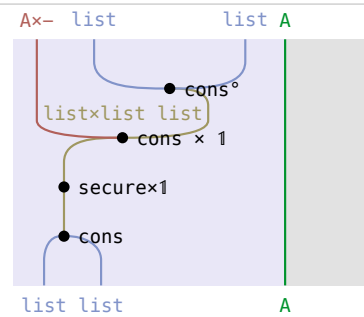


`secure prefix⊑prefix secure`

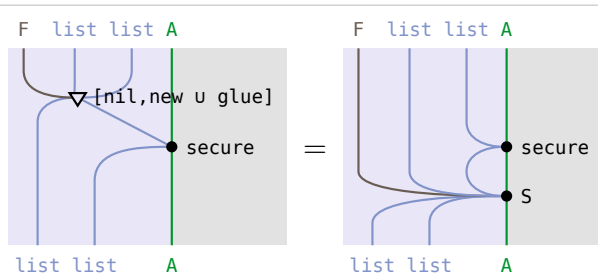
every pair `secure` then `prefix` gives, `prefix` then `secure` gives too — a prefix of a secure stretch is itself secure, which is the prefix-closure B&dM p.185 names

§7.5.2. `partition list(secure)=⦇[nil,new ∪ old]⦈`

(7.5.2a)

definition	Hinze–Marsden
`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	
`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	
`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)

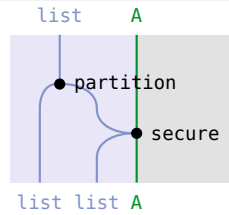
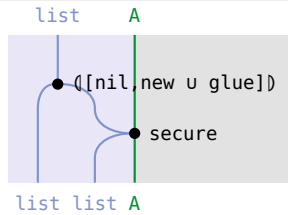
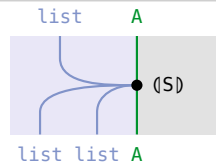


`(∀a. secureP(amount,N,[a])) ⟹ [nil,new ∪ glue]list(secure)=F(list(secure))S`

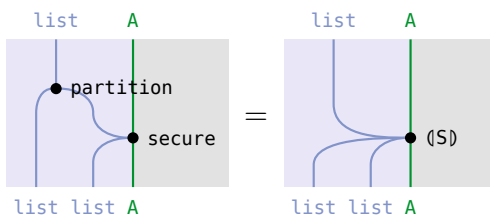
testing every segment for security after the algebra has run is testing it before it ran: a segment `glue` has extended is secure exactly where `old` would have let it be, and the one transaction `new` sets out on its own is secure whatever it is

(∀a. secureP(amount,N,[a])) ⟹ partition list(secure)=⦇S⦈ (7.5.2c)

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`	

(7.5.2d)



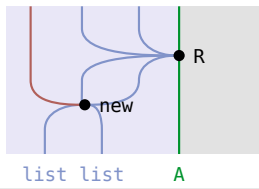
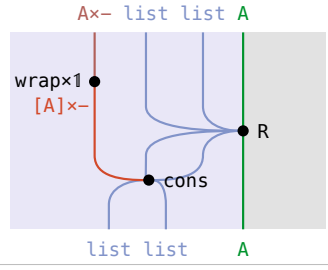
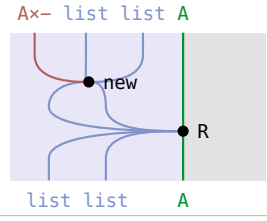
`(∀a. secureP(amount,N,[a])) ⟹ partition list(secure)=⦇S⦈`

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`

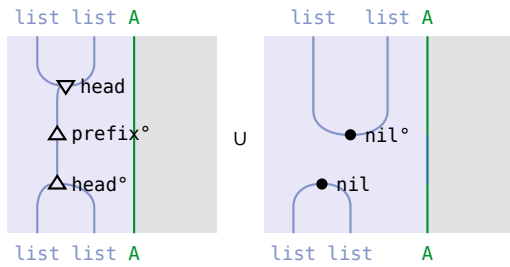
(𝟙×R)new⊑(new ∪ old)R (7.5.3a)

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) — ≜ `secure`	
= `(wrap×R)cons` `new≜(wrap×𝟙)cons` — (7.5.2a) — 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)`

(7.5.4a)

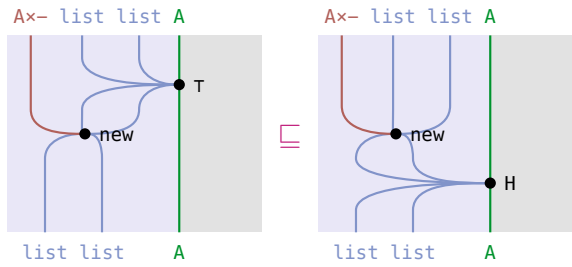


H≜head prefix°head° ∪ nil°nil

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)

(7.5.5a)

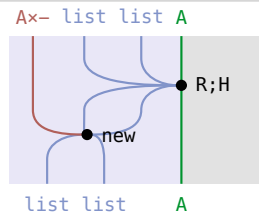
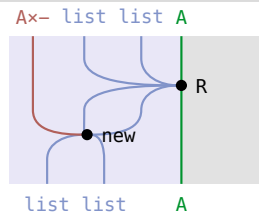
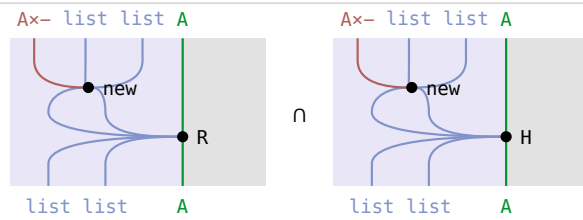
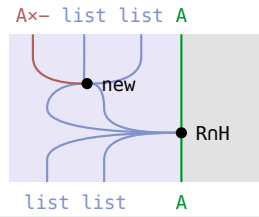
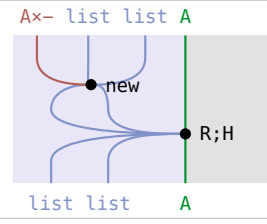


(𝟙×T)new⊑new H

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

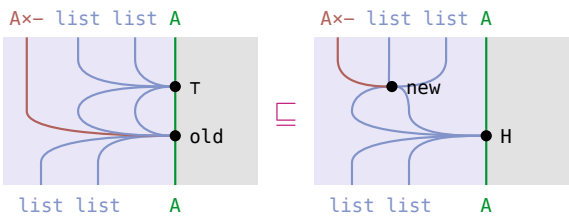
(𝟙×(R;H))new⊑(new ∪ old)(R;H) (7.5.5b)

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 — ≜ secure	
⊑ new R ∩ new H (7.14) as far as its new R line — (7.5.3a) — 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)

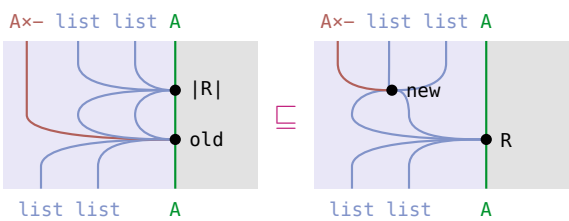
(7.5.6a)



`(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

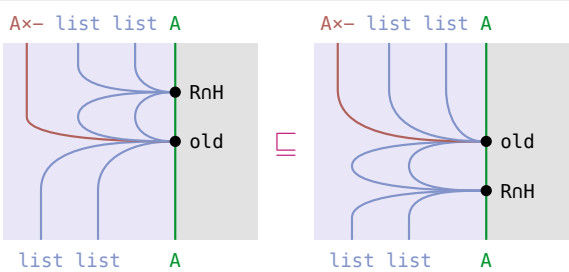
(7.5.6b)



`(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

(7.5.6c)

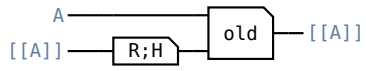
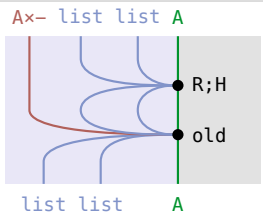
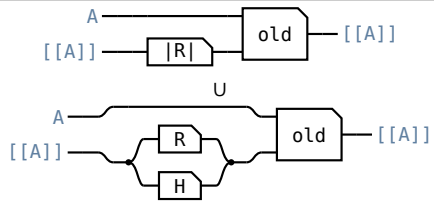
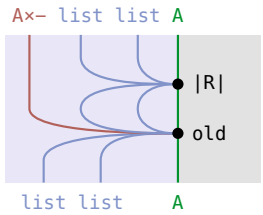
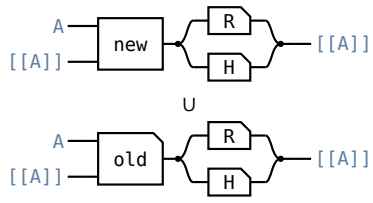
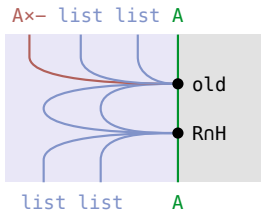
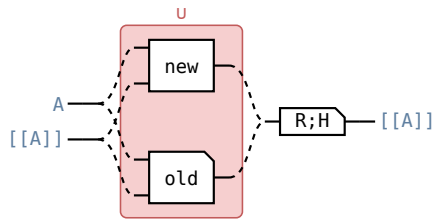
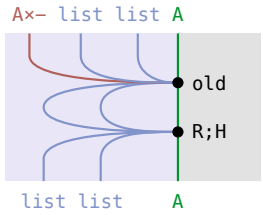


`(𝟙×(R∩H))old⊑old (R∩H)`

on a tie the one first segment is a prefix of the other, so prefix-closure of `secure` — (7.5.1b) — lets `old` fire on this side too, and it keeps both the length and the prefix

(𝟙×(R;H))old⊑(new ∪ old)(R;H) (7.5.6d)

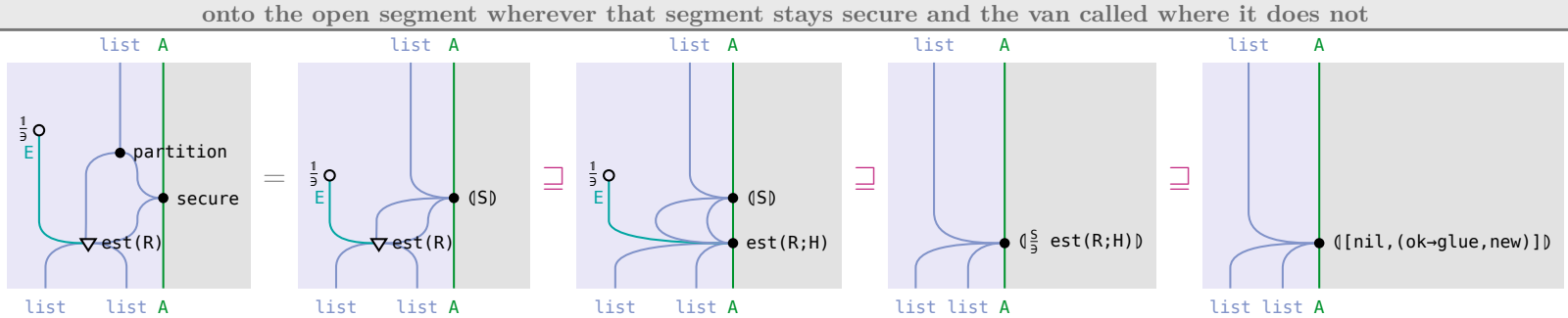
gluing the transaction onto a better schedule for the rest gets no further than gluing it on, or calling the van, and bettering the whole schedule after,

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)` — ≜ `secure`, ∪ distributes,	
⊑  `new (R∩H) ∪ old (R∩H)` (7.19) and (7.20) on `|R|` — (7.5.6a), (7.5.6b) — and (7.21) on `R∩H` — (7.5.6c)	
⊑  `(new ∪ old)(R;H)` `X∩Y⊑X;Y`, converses	

§7.5.7. The derivation

(∀a. secureP(amount,N,[a])) ⟹ ⦇[nil,(ok→glue,new)]⦈⊑$^{\text{partition list(secure)}}_{\ni}$ 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



(a) `partition list(secure)=⦇S⦈` — ≜ `secure`, fusion

(b) greedy theorem at `R;H` — (7.5.6d), (7.5.5b), (7.5.5a)

§8. Thinning Algorithms

§8.1. Thinning

(8.1a)

For `Q : A→A`, `thin(Q)≜(∋/∋)∩(∈\(Q°∈)) : EA→EA` (8.1).

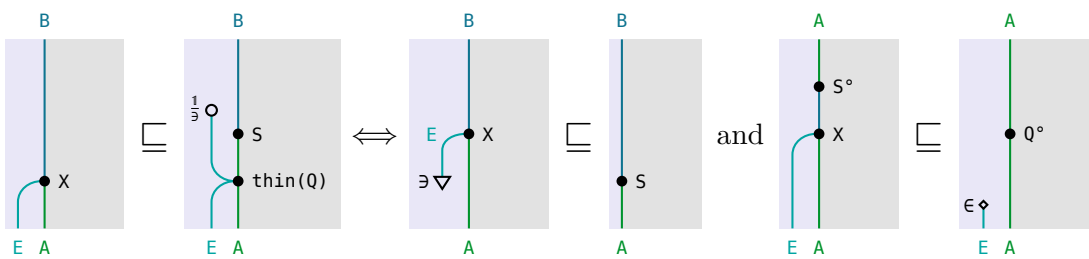
`ys thin(Q) xs⟺xs⊆ys∧(∀a∈ys. ∃b∈xs. b Q a)` the same ° as ≜ `est`: `∈X` runs `a→ys→xs`, member of `ys` first, and `Q°∈` runs `a→b→xs`, so `(a,b)∈Q°` reads `b Q a`; at `Q≜≤` every `a∈ys` keeps some `b≤a` in `xs`, the end `est(≤)` picks

(8.1b)

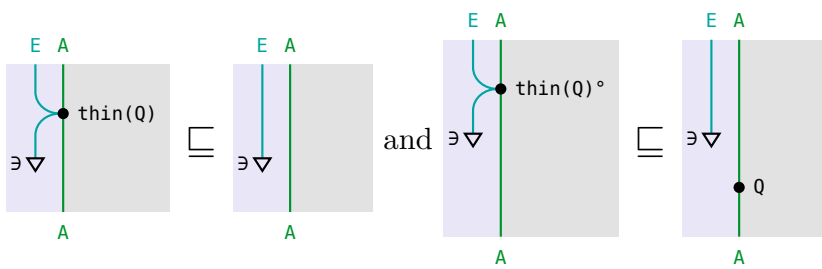
the law	what it says
`X⊑ S/∋ thin(Q)⟺X∋⊑S` and `S°X⊑Q°∈`	everything kept is an `S`-value, and every `S`-value has a `Q`-lower bound among the kept ones
`thin(Q)∋⊑∋`	everything a thinning keeps was in the set
`thin(Q)°∋⊑∋Q`	every element of the set has a `Q`-lower bound among the kept ones
`Q⊑R ⟹ thin(Q)⊑thin(R)`	the fewer pairs `Q` relates, the fewer subsets count as thinnings
`𝟙⊑Q ⟹ 𝟙⊑thin(Q)` `QQ⊑Q ⟹ thin(Q)thin(Q)⊑thin(Q)`	keeping everything is always a legal thinning
`Q⊑R ∧ 𝟙⊑Q ∧ RR⊑R ⟹ est(R)=thin(Q)est(R)` `Q⊑R`, `𝟙⊑Q`, `RR⊑R` — (8.1.2a); weaker than the book's "both preorders": `Q` transitive is never used and `𝟙⊑R` follows	**thin-introduction**: thinning first cannot lose an `R`-minimum
`thin(Q)⊒est(Q) 𝟙/∋` (8.2) — (8.1.2d)	**thin-elimination**: keeping one element is a thinning, but its domain is the sets `est(Q)` is defined on
`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
`P(thin(Q))union⊑union thin(Q)` (8.4)	thinning each member set is a thinning of the union

§8.1.1. `X⊑(S%∋) thin(Q)⟺X∋⊑S` and `S°X⊑Q°∈`

(8.1.1a)



(8.1.1b)

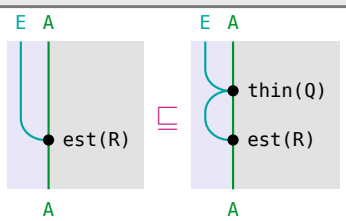


(8.1.1a) at `X≜thin(Q)`, `S≜∋`: the left side is `thin(Q)⊑thin(Q)`, since `∋%∋=Λ(∋)=𝟙`

§8.1.2. `est(R)=thin(Q) est(R)` given `Q⊑R, 𝟙⊑Q, RR⊑R`

est(R)⊑thin(Q) est(R) (8.1.2a)

the ⊑ half: keeping everything is a thinning — 𝟙⊑Q

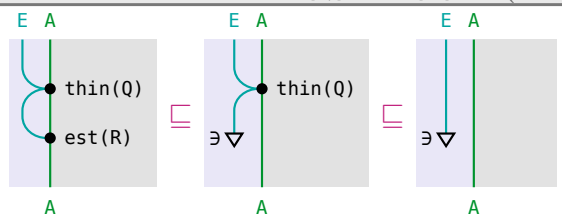


(a)

(a) 𝟙⊑thin(Q) — (8.1b)

thin(Q) est(R)⊑∋ (8.1.2b)

the ∋ half, first condition of the UP of est at X≜thin(Q) est(R) — (7.1.1a)

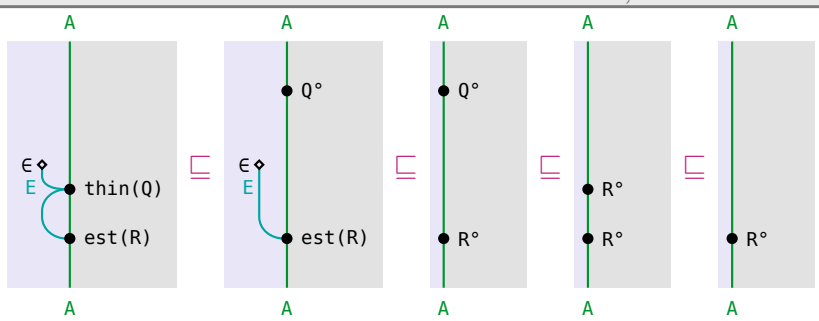


(a) (b)

(a) est(R)⊑∋ — (7.1b)
(b) thin(Q)∋⊑∋ — (8.1.1a)

∈ thin(Q) est(R)⊑R° (8.1.2c)

the ∋ half, second condition — Q⊑R, R transitive



(a) (b) Q⊑R (c)

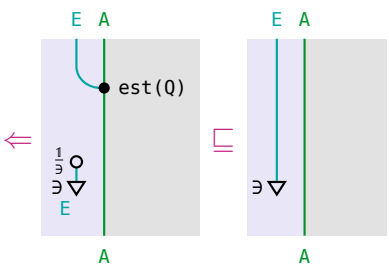
(a) ∈ thin(Q)⊑Q°∈ — (8.1.1a)
(b) ∈ est(R)⊑R° — (7.1b)
(c) R transitive

est(Q)$\frac{1}{∋}$⊑thin(Q) (8.1.2d)

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

est(Q)$\frac{1}{∋}$∋⊑∋

every member of the singleton is a member of the set

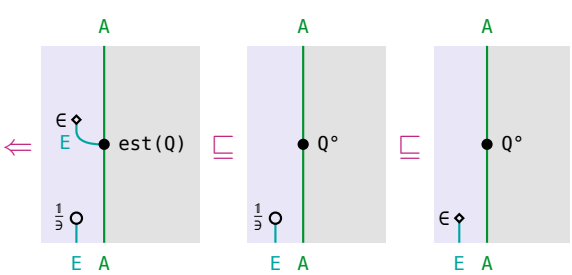


(a) (b)

(a) ≜ thin, ∋/∋ half
(b) $\frac{1}{∋}$∋=𝟙 — (3.1j); est(Q)⊑∋ — (7.1b)

∈est(Q)$\frac{1}{∋}$⊑Q°∈

the singleton's member stands in Q to every member of the set



(a) (b) (c)

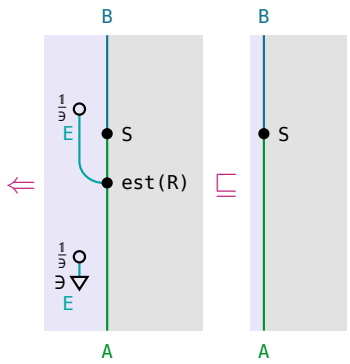
(a) ≜ thin, ∈\(Q°∈) half
(b) ∈ est(Q)⊑Q° — (7.1.1a)
(c) $\frac{1}{∋}$⊑∈ — (3.1j)

$$\mathsf{R}\cap\mathsf{S}^\circ\mathsf{S}\sqsubseteq\mathsf{Q} \implies \tfrac{\mathsf{S}}{\ni}\ \mathsf{est}(\mathsf{R})\tfrac{1}{\ni}\sqsubseteq\tfrac{\mathsf{S}}{\ni}\ \mathsf{thin}(\mathsf{Q}) \qquad (8.1.2e)$$

given $\mathsf{R}\cap(\mathsf{S}^\circ\mathsf{S})\sqsubseteq\mathsf{Q}$, Q a preorder

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

every member of the singleton is a value S returns



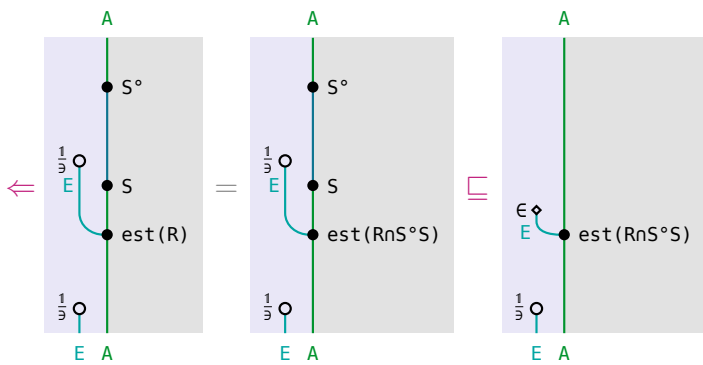
(a) (b)

(a) (8.1b), first condition

(b) $\tfrac{1}{\ni}\ni=\mathbb{1}$ — (3.1j); $\tfrac{\mathsf{S}}{\ni}\ \mathsf{est}(\mathsf{R})\sqsubseteq\mathsf{S}$ — (7.1b)

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

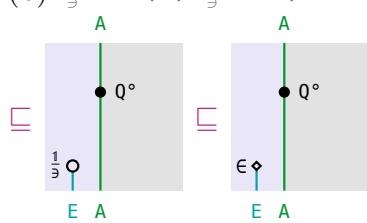
the singleton's member stands in Q to every value S returns at the same argument



(a) (b) $\mathsf{S}^\circ\tfrac{\mathsf{S}}{\ni}\sqsubseteq\in$ — (3.1j)

(a) (8.1b), second condition

(b) $\tfrac{\mathsf{S}}{\ni}\ \mathsf{est}(\mathsf{R})=\tfrac{\mathsf{S}}{\ni}\ \mathsf{est}(\mathsf{R}\cap\mathsf{S}^\circ\mathsf{S})$ — (7.1b)



(a) (b)

(a) $\in\ \mathsf{est}(\mathsf{R}\cap\mathsf{S}^\circ\mathsf{S})\sqsubseteq(\mathsf{R}\cap\mathsf{S}^\circ\mathsf{S})^\circ$ — (7.1.1a); $\mathsf{R}\cap(\mathsf{S}^\circ\mathsf{S})\sqsubseteq\mathsf{Q}$

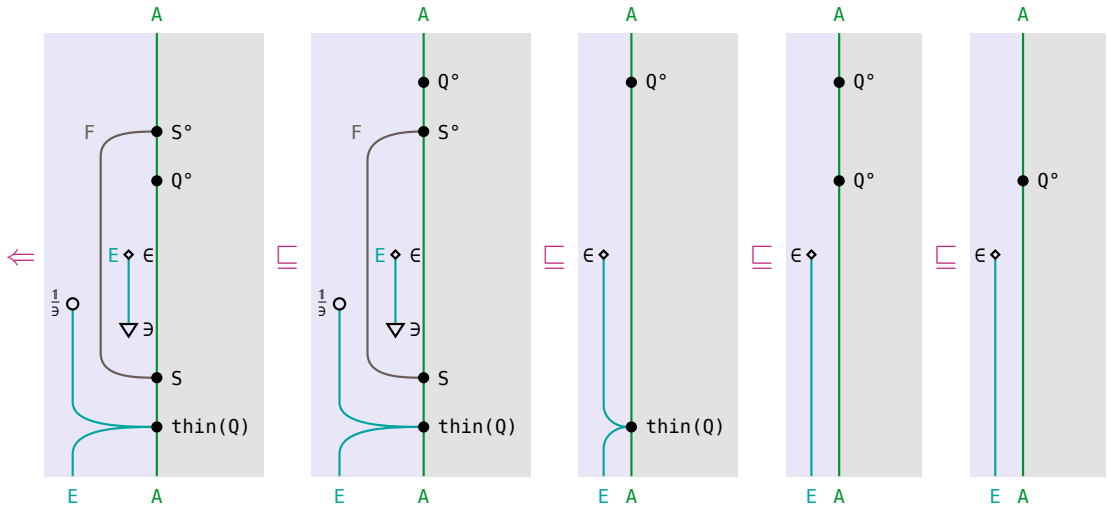
(b) $\tfrac{1}{\ni}\sqsubseteq\in$ — (3.1j)

$$\mathtt{QQ{\sqsubseteq}Q} \wedge \mathtt{MonoAlg(S,Q)} \implies (\!|\tfrac{F(\ni)S}{\ni}\ \mathtt{thin(Q)}|\!){\sqsubseteq}\tfrac{(\!|S|\!)}{\ni}\ \mathtt{thin(Q)} \qquad (8.1.2f)$$

thinning at every step of the reduce is a thinning of the whole candidate set — **S** monotonic on **Q**, **Q** a preorder

$(\!|\tfrac{F(\ni)S}{\ni}\ \mathtt{thin(Q)}|\!)\ni{\sqsubseteq}(\!|S|\!)$ and $(\!|S|\!)^\circ(\!|\tfrac{F(\ni)S}{\ni}\ \mathtt{thin(Q)}|\!){\sqsubseteq}Q^\circ\in$

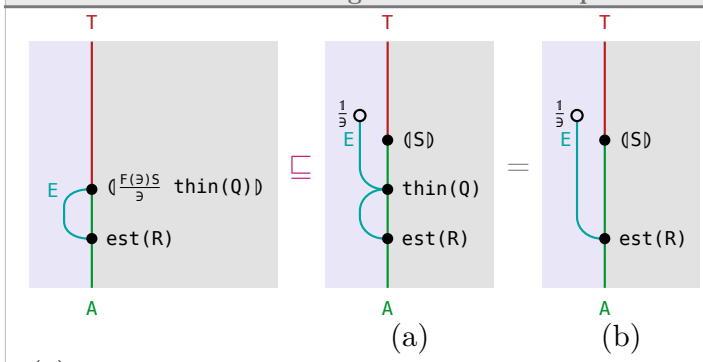
(8.1b) at $X{\triangleq}(\!|\tfrac{F(\ni)S}{\ni}\ \mathtt{thin(Q)}|\!)$, $(\!|S|\!)$ for its S



(a) fusion, hylomorphism theorem
(b) $S^\circ F(Q^\circ){\sqsubseteq}Q^\circ S^\circ$ — (7.2b), (5.1b)
(c) $S^\circ F(\in)\frac{F(\ni)S}{\ni}{\sqsubseteq}\in$ — (3.1j)
(d) $\in\ \mathtt{thin(Q)}{\sqsubseteq}Q^\circ\in$ — $\triangleq$ thin, (3.1a)

$$\mathtt{Q{\sqsubseteq}R} \wedge \mathtt{1{\sqsubseteq}Q} \wedge \mathtt{QQ{\sqsubseteq}Q} \wedge \mathtt{RR{\sqsubseteq}R} \wedge \mathtt{MonoAlg(S,Q)} \implies (\!|\tfrac{F(\ni)S}{\ni}\ \mathtt{thin(Q)}|\!)\mathtt{est(R)}{\sqsubseteq}\tfrac{(\!|S|\!)}{\ni}\ \mathtt{est(R)} \qquad (8.1.2g)$$

the thinning fold refines the optimisation problem itself — **S** monotonic on **Q**, **Q⊑R**, both preorders



(a) thinning theorem
(b) est(R)=thin(Q) est(R) — (8.1b)

§8.2. Paths in a layered network

(8.2a)

F(A,X)=A+A×X, L=list⁺ with initial algebra $\alpha{\triangleq}$[wrap,cons] : F(A,LA)→LA.

wrapz≜⟨wrap,zero⟩, consw(a,(xs,n))=(cons(a,xs),wt(a,head(xs))+n).

cost≜(|[wrapz,consw]|)π₂, (|[wrapz,consw]|)=⟨1,cost⟩, R≜cost≤cost°.

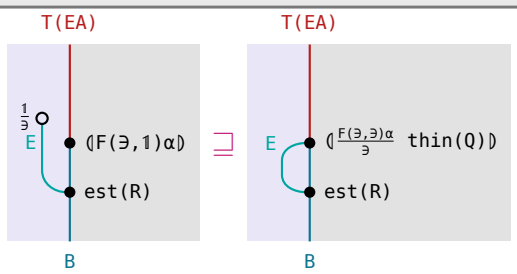
Q≜Rn(head head°), S≜F(1,∋)α, $\frac{F(\ni,1)}{\ni}$ =1+cpl, $\frac{F(1,\ni)}{\ni}$ =1+cpr, step≜cpr P(cons) est(R).

(8.2b)

the law	what it says
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
S head⊑[1,π₁] S≜F(1,∋)α	S head is simple, which gives Rn(S°S)⊑Q: between two paths S builds from one argument, equal cost and equal head already means Q

$$Q \sqsubseteq R \wedge \mathbb{1} \sqsubseteq Q \wedge QQ \sqsubseteq Q \wedge R^\circ R^\circ \sqsubseteq R^\circ \wedge \mathrm{MonoAlg}(F(\ni,\mathbb{1})\alpha,Q) \wedge R\cap(F(\mathbb{1},\ni)\alpha)^\circ F(\mathbb{1},\ni)\alpha \sqsubseteq Q \implies (\!|\tfrac{F(\ni,\mathbb{1})}{\ni}\ P(\tfrac{F(\mathbb{1},\ni)\alpha}{\ni}\ \mathrm{est}(R))|\!)\mathrm{est}(R) \sqsubseteq \tfrac{(\!|F(\ni,\mathbb{1})\alpha|\!)}{\ni}\ \mathrm{est}(R) \qquad (8.2c)$$

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



$= \frac{L(\ni)}{\ni}\ \mathrm{est}(R)$ (8.1.2g), (8.2b)

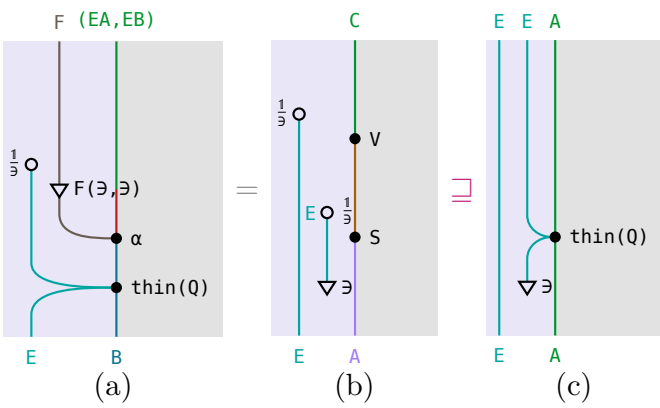
$\sqsupseteq$ (8.2d) under $(\!|\ |\!)$ monotonic: the whole chain runs inside the reduce, and the $\mathrm{est}(R)$ behind it never moves.

$$R\cap(F(\mathbb{1},\ni)\alpha)^\circ F(\mathbb{1},\ni)\alpha \sqsubseteq Q \implies \tfrac{F(\ni,\mathbb{1})}{\ni}\ P(\tfrac{F(\mathbb{1},\ni)\alpha}{\ni}\ \mathrm{est}(R)) \sqsubseteq \tfrac{F(\ni,\ni)\alpha}{\ni}\ \mathrm{thin}(Q) \qquad (8.2d)$$

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

$$\tfrac{F(\ni,\mathbb{1})}{\ni}\ P(\tfrac{F(\mathbb{1},\ni)\alpha}{\ni})P(\mathrm{thin}(Q))\mathrm{union} \sqsubseteq \tfrac{F(\ni,\ni)\alpha}{\ni}\ \mathrm{thin}(Q)$$

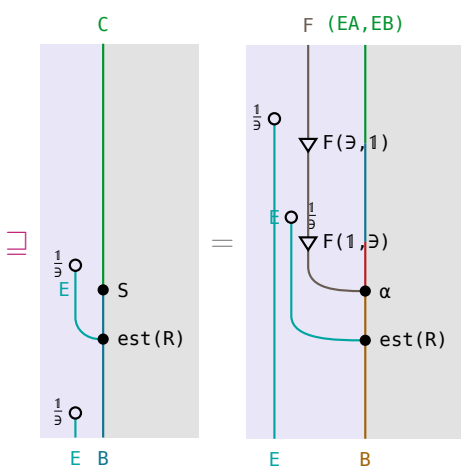
thinning each set the algebra's parts build, then taking the union, is below thinning the one set the whole algebra builds



(a) the algebra of (8.2c) row 2
(b) $\frac{F(\ni,\mathbb{1})F(\mathbb{1},\ni)\alpha}{\ni} = \frac{F(\ni,\mathbb{1})}{\ni}\ P(\frac{F(\mathbb{1},\ni)\alpha}{\ni})\ \mathrm{union}$
(c) $\mathrm{union}\ \mathrm{thin}(Q) \sqsupseteq P(\mathrm{thin}(Q))\ \mathrm{union}$ — (8.1b)

$$R\cap(F(\mathbb{1},\ni)\alpha)^\circ F(\mathbb{1},\ni)\alpha \sqsubseteq Q \implies \tfrac{F(\ni,\mathbb{1})}{\ni}\ P(\tfrac{F(\mathbb{1},\ni)\alpha}{\ni}\ \mathrm{est}(R)) \sqsubseteq \tfrac{F(\ni,\mathbb{1})}{\ni}\ P(\tfrac{F(\mathbb{1},\ni)\alpha}{\ni})P(\mathrm{thin}(Q))\mathrm{union}$$

choosing an R-least member of each of those sets is below thinning each and taking the union

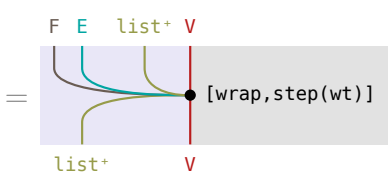


$P(\frac{\mathbb{1}}{\ni})\ \mathrm{union}=\mathbb{1}$

(a) $\frac{S}{\ni}\ \mathrm{thin}(Q) \sqsupseteq \frac{S}{\ni}\ \mathrm{est}(R)\ \frac{\mathbb{1}}{\ni}$ — (8.1b), (8.2b)

$$\mathrm{cp}(F,\mathrm{list}^+(V))P(\alpha)\mathrm{est}(\mathrm{path}(wt))=[\mathrm{wrap},\mathrm{step}(wt)]$$

at the layered network, choosing an R-least path from each case is the program's two cases $[\mathrm{wrap},\mathrm{step}]$



(a) α a map; $\frac{F(\mathbb{1},\ni)}{\ni}\ P(\alpha)\ \mathrm{est}(R)=[\mathrm{wrap},\mathrm{step}]$ — $\triangleq \mathrm{cost}$

$= \frac{F(\ni,\mathbb{1})}{\ni}\ =\mathbb{1}+\mathrm{cpl}$ — $\triangleq \mathrm{cost}$

§8.3. Implementing thin

(8.3a)

`setify : [A]→EA`, `cup : EA×EA→EA`, `cp(F)≜` $\frac{\mathtt{F(\ni)}}{\ni}$, `listcp : F(L)→LF`, `sort(P)≜setify° ordered P` for `P` a connected preorder.

`thinlist(Q)` is any `thinlist(Q)⊑subseq` with `thinlist(Q) setify⊑setify thin(Q)`; one is `⦇[nil,bump Q]⦈`, `bump Q (a,[])=[a]`, `bump Q (a,[b]♯xs)=(b Q a→[a]♯xs,a Q b→[b]♯xs,[a]♯[b]♯xs)`.

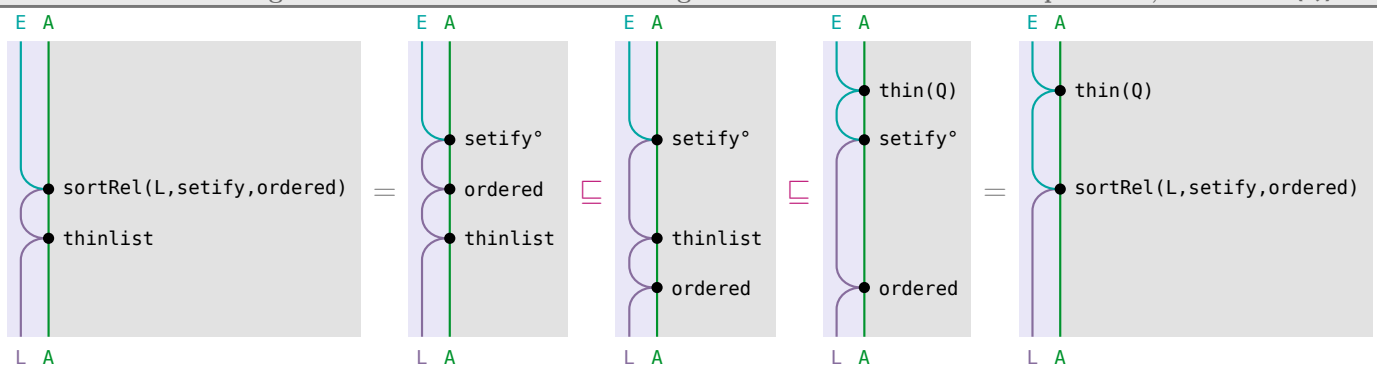
Binary thinning data: `S=(f₁p₁) ∪ (f₂p₂)` with `p₁`, `p₂` coreflexive; `Q` a preorder with `Q⊑R` and both `f₁p₁`, `f₂p₂` monotonic on `Q`; `P` a connected preorder with both `f₁`, `f₂` monotonic on `P`; `gᵢ≜list(fᵢ) filter(pᵢ)`.

(8.3b)

the law	what it says
`thinlist(Q) xs=[minlist(Q) xs]` (8.5), `Q` connected, `xs` non-empty	what thinning should come to when it can: one element
`sort(P) thinlist(Q)⊑thin(Q) sort(P)` (8.6) — (8.3c)	thinning a sorted list is a thinning of the set — this is what `thinlist(Q)⊑subseq` buys
`sort(P) minlist(Q)⊑est(Q)` (8.7)	a minimum of the sorted list is a minimum of the set
`sort(fPf°) list(f)⊑P(f) sort(P)` (8.8)	shunt a function through a sort
`sort(P) filter(p)⊑E(p) sort(P)` (8.9), `p` coreflexive	filtering a sorted list sorts the restricted set
`(sort(P)×sort(P)) merge(P)⊑cup sort(P)` (8.10)	merging two sorted lists sorts their union
`F(sort(P)) listcp⊑cp(F) sort(FP)` (8.11), `F` linear	`listcp` is the list implementation of the cartesian product `cp(F)`

`Map(setify) ∧ Coreflexive(ordered) ∧ thinlist⊑subseq ∧ ordered subseq⊑subseq ordered ∧ thinlist setify⊑setify thin(Q) ⟹ sortRel(L,setify,ordered)thinlist⊑ thin(Q)sortRel(L,setify,ordered)` (8.3c)

a thinning of the sorted list lists a thinning of the set — `P` a connected preorder, `thinlist(Q)⊑subseq`.



(a) (b) (c) (d)

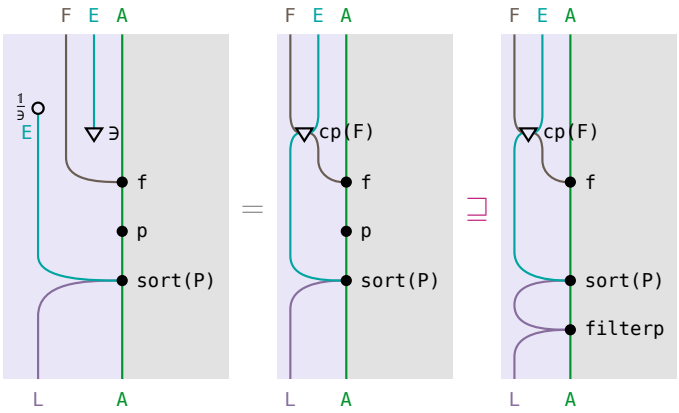
(a) `sort(P)≜setify° ordered P` — ≜ `thinlist`
(b) `ordered P thinlist(Q)⊑thinlist(Q) ordered P` — ≜ `thinlist`
(c) `thinlist(Q) setify⊑setify thin(Q)` — ≜ `thinlist`, (3.1g), (3.1c)
(d) `sort(P)≜setify° ordered P` — ≜ `thinlist`

Map(f) ∧ (∀X. ∀Y. X⊑Y ⟹ sortF(X)⊑sortF(Y)) ∧ MonoAlg(f,P) ∧ sortF(fPf°)listf⊑P(f)sort(P) ∧ sort(P)filterp⊑E(p)sort(P) ∧ F(sort(P))listcp⊑cp(F,A)sortF(F(P)) ⟹ F(sort(P))listcp listf filterp⊑$\frac{F(\ni)fp}{\ni}$ sort(P) (8.3d)

one sorted list built from sorted arguments, instead of a set built and then sorted — **f : FA→A** monotonic on **P**, **p** coreflexive, **F** linear.

Map(f) ∧ sort(P)filterp⊑E(p)sort(P) ⟹ cp(F,A)P(f)sort(P)filterp⊑$\frac{F(\ni)fp}{\ni}$ sort(P)

sorting the f-images of the product and filtering by p is below sorting the set F(∋)fp builds

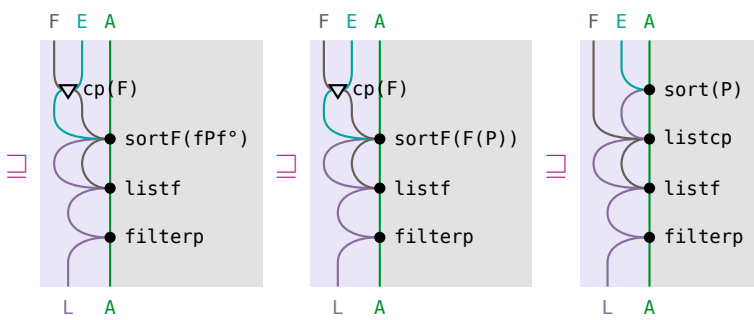


(a) (b)

(a) $\frac{F(\ni)fp}{\ni} = \frac{F(\ni)}{\ni}$ E(fp) — (3.1j), ≜ thinlist
E(f)=P(f), f a map — (5.4d)
(b) sort(P) filter(p)⊑E(p) sort(P) — (8.3b)

Map(f) ∧ (∀X. ∀Y. X⊑Y ⟹ sortF(X)⊑sortF(Y)) ∧ MonoAlg(f,P) ∧ sortF(fPf°)listf⊑P(f)sort(P) ∧ F(sort(P))listcp⊑cp(F,A)sortF(F(P)) ⟹ F(sort(P))listcp listf filterp⊑cp(F,A)P(f)sort(P)filterp

combining the arguments' sorted lists by listcp, mapping f and filtering is below sorting the f-images and filtering

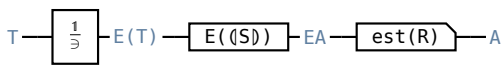
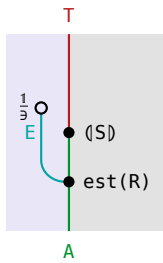
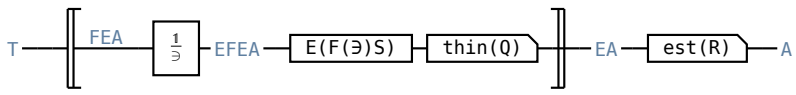
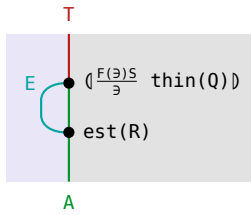
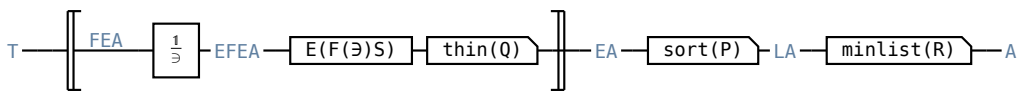
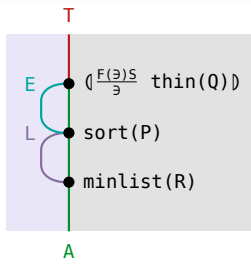
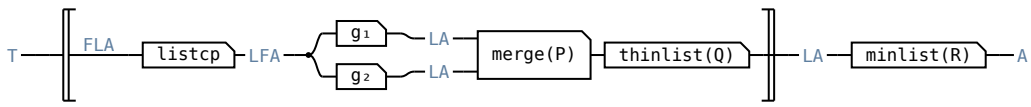
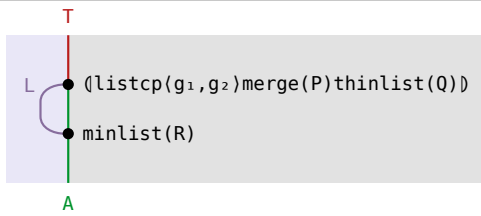


(a) (b) (c)

(a) sort(fPf°) list(f)⊑P(f) sort(P) — (8.3b)
(b) FP⊑fPf° — (7.2b), ≜ thinlist
(c) F(sort(P)) listcp⊑cp(F) sort(FP) — (8.3b)

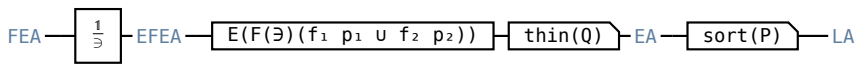
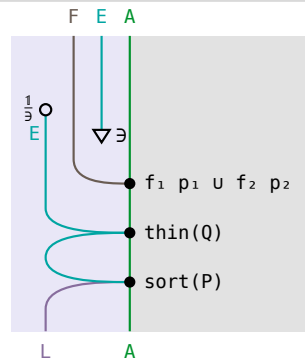
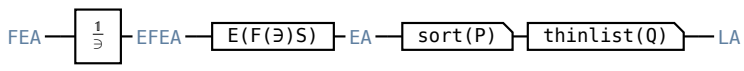
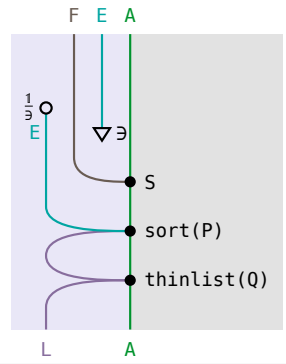
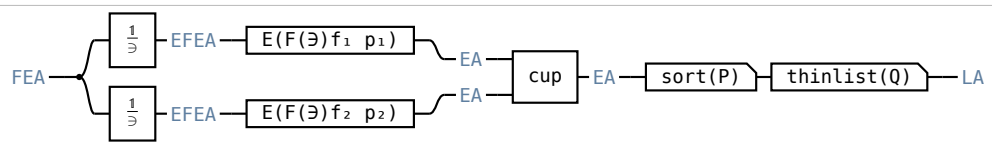
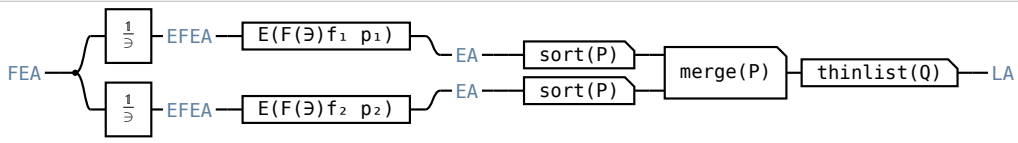
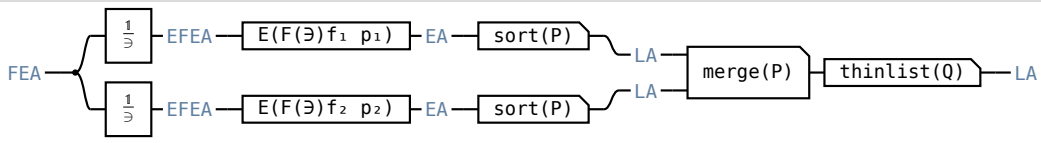
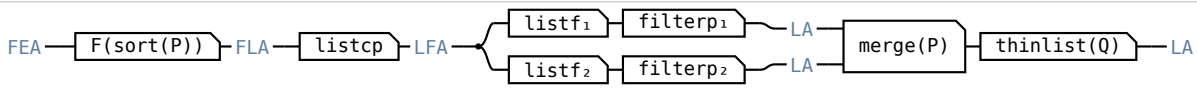
$$\mathtt{Map}(f_1) \land \mathtt{Map}(f_2) \land Q \sqsubseteq R \land \mathbb{1} \sqsubseteq Q \land QQ \sqsubseteq Q \land R^\circ R^\circ \sqsubseteq R^\circ \land \mathtt{MonoAlg}(f_1\ p_1, Q) \land \mathtt{MonoAlg}(f_2\ p_2, Q) \land (\forall X.\ \forall Y.\ X \sqsubseteq Y \Longrightarrow \mathtt{sortF}(X) \sqsubseteq \mathtt{sortF}(Y)) \land \mathtt{MonoAlg}(f_1, P) \land \mathtt{MonoAlg}(f_2, P) \land \mathtt{sortF}(f_1\ Pf_1^\circ)\mathtt{listf}_1 \sqsubseteq P(f_1)\mathtt{sort}(P) \land \mathtt{sortF}(f_2\ Pf_2^\circ)\mathtt{listf}_2 \sqsubseteq P(f_2)\mathtt{sort}(P) \land \mathtt{sort}(P)\mathtt{filterp}_1 \sqsubseteq E(p_1)\mathtt{sort}(P) \land \mathtt{sort}(P)\mathtt{filterp}_2 \sqsubseteq E(p_2)\mathtt{sort}(P) \land F(\mathtt{sort}(P))\mathtt{listcp} \sqsubseteq \mathtt{cp}(F, A)\mathtt{sortF}(F(P)) \land (\mathtt{sort}(P) \times \mathtt{sort}(P))\mathtt{merge}(P) \sqsubseteq \mathtt{cup}\ \mathtt{sort}(P) \land \mathtt{sort}(P)\mathtt{thinlist}(Q) \sqsubseteq \mathtt{thin}(Q)\mathtt{sort}(P) \land \mathtt{sort}(P)\mathtt{minlist}(R) \sqsubseteq \mathtt{est}(R) \land g_1 = \mathtt{listf}_1\ \mathtt{filterp}_1 \land g_2 = \mathtt{listf}_2\ \mathtt{filterp}_2 \land S = f_1\ p_1 \cup f_2\ p_2 \Longrightarrow (\!|\mathtt{listcp}(g_1, g_2)\mathtt{merge}(P)\mathtt{thinlist}(Q)|\!)\mathtt{minlist}(R) \sqsubseteq \frac{(\!|S|\!)}{\ni}\ \mathtt{est}(R) \qquad (8.3e)$$

a fold on sorted lists of partial solutions, thinned at every step — at ≜ **thinlist**'s binary thinning data.

circuit	Hinze–Marsden
	
⊑  (8.1.2g) at $f_1 p_1$ and $f_2 p_2$ monotonic on Q — ≜ thinlist	
⊑  $\mathtt{sort}(P)\ \mathtt{minlist}(R) \sqsubseteq \mathtt{est}(R)$ — (8.3b) at its $Q \triangleq R$	
⊑  fusion at (8.3f)	

$$\mathtt{Map}(f_1) \land \mathtt{Map}(f_2) \land (\forall X.\ \forall Y.\ X \sqsubseteq Y \Longrightarrow \mathtt{sortF}(X) \sqsubseteq \mathtt{sortF}(Y)) \land \mathtt{MonoAlg}(f_1,P) \land \mathtt{MonoAlg}(f_2,P) \land \mathtt{sortF}(f_1\ Pf_1{}^\circ)\mathtt{listf}_1 \sqsubseteq P(f_1)\mathtt{sort}(P) \land \mathtt{sortF}(f_2\ Pf_2{}^\circ)\mathtt{listf}_2 \sqsubseteq P(f_2)\mathtt{sort}(P) \land \mathtt{sort}(P)\mathtt{filterp}_1 \sqsubseteq E(p_1)\mathtt{sort}(P) \land \mathtt{sort}(P)\mathtt{filterp}_2 \sqsubseteq E(p_2)\mathtt{sort}(P) \land F(\mathtt{sort}(P))\mathtt{listcp} \sqsubseteq \mathtt{cp}(F,A)\mathtt{sortF}(F(P)) \land (\mathtt{sort}(P)\times\mathtt{sort}(P))\mathtt{merge}(P) \sqsubseteq \mathtt{cup}\ \mathtt{sort}(P) \land \mathtt{sort}(P)\mathtt{thinlist}(Q) \sqsubseteq \mathtt{thin}(Q)\mathtt{sort}(P) \Longrightarrow F(\mathtt{sort}(P))\mathtt{listcp}\langle \mathtt{listf}_1\ \mathtt{filterp}_1, \mathtt{listf}_2\ \mathtt{filterp}_2\rangle\ \mathtt{merge}(P)\mathtt{thinlist}(Q) \sqsubseteq \frac{F(\ni)(f_1\ p_1 \cup f_2\ p_2)}{\ni}\ \mathtt{thin}(Q)\mathtt{sort}(P) \tag{8.3f}$$

sorting the candidate set is what turns the thinning algebra into an algebra on lists — the side condition of binary thinning theorem's last step.

circuit	Hinze–Marsden
	
⊑  $\mathtt{sort}(P)\ \mathtt{thinlist}(Q) \sqsubseteq \mathtt{thin}(Q)\ \mathtt{sort}(P)$ — (8.3b)	
=  $\langle \frac{F(\ni)f_1p_1}{\ni}, \frac{F(\ni)f_2p_2}{\ni}\rangle\ \mathtt{cup}\ \mathtt{sort}(P)\ \mathtt{thinlist}(Q)$ $S=(f_1p_1) \cup (f_2p_2)$ — $\triangleq$ thinlist, then $\triangleq$ cup	
⊑  $\langle \frac{F(\ni)f_1p_1}{\ni}, \frac{F(\ni)f_2p_2}{\ni}\rangle(\mathtt{sort}(P)\times\mathtt{sort}(P))\ \mathtt{merge}(P)\ \mathtt{thinlist}(Q)$ $(\mathtt{sort}(P)\times\mathtt{sort}(P))\ \mathtt{merge}(P) \sqsubseteq \mathtt{cup}\ \mathtt{sort}(P)$ — (8.3b)	
=  $\langle \frac{F(\ni)f_1p_1}{\ni}\ \mathtt{sort}(P), \frac{F(\ni)f_2p_2}{\ni}\ \mathtt{sort}(P)\rangle\ \mathtt{merge}(P)\ \mathtt{thinlist}(Q)$ $\langle X,Y\rangle(\mathtt{sort}(P)\times\mathtt{sort}(P)) = \langle X\ \mathtt{sort}(P), Y\ \mathtt{sort}(P)\rangle$ — (5.2.2b)	
⊑  $F(\mathtt{sort}(P))\ \mathtt{listcp}\ \langle g_1,g_2\rangle\ \mathtt{merge}(P)\ \mathtt{thinlist}(Q)$ (8.3d) at f_1, p_1 and at f_2, p_2, then $X\langle g_1,g_2\rangle \sqsubseteq \langle Xg_1,Xg_2\rangle$ — (5.2.2b); $g_i \triangleq \mathtt{list}(f_i)\ \mathtt{filter}(p_i)$ — $\triangleq$ thinlist	

§8.4. The knapsack problem

(8.4a)

$\mathtt{vol},\mathtt{wt} : \mathtt{Item} \to \mathtt{Real},\ \mathtt{value} \triangleq \mathtt{list}(\mathtt{vol})\ \mathtt{sum},\ \mathtt{weight} \triangleq \mathtt{list}(\mathtt{wt})\ \mathtt{sum}$.

$\mathtt{subseq} = (\!|[\mathtt{nil},\mathtt{cons}] \cup [\mathtt{nil},\pi_2]|\!)$, , within w the coreflexive on $\mathtt{xs}$ with $\mathtt{weight}\ \mathtt{xs} \le w$, $0 \le w$.

$R \triangleq \mathtt{value} \ge \mathtt{value}^\circ$, $Q \triangleq R \cap (\mathtt{weight} \le \mathtt{weight}^\circ)$, $P \triangleq R$, .

$FA = 1 + \mathtt{Item} \times A$, $\mathtt{listcp} = \mathtt{wrap} + \mathtt{cpr}$, $g_1 \triangleq \mathtt{list}([\mathtt{nil},\mathtt{cons}])\ \mathtt{filter}(\mathtt{within}\ w) = [\mathtt{list}(\mathtt{nil}), h_1]$, $g_2 \triangleq \mathtt{list}([\mathtt{nil},\pi_2]) = [\mathtt{list}(\mathtt{nil}), h_2]$.

$h_1 \triangleq \mathtt{list}(\mathtt{cons})\ \mathtt{filter}(\mathtt{within}\ w)$, $h_2 \triangleq \mathtt{list}(\pi_2)$.

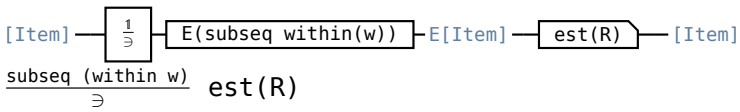
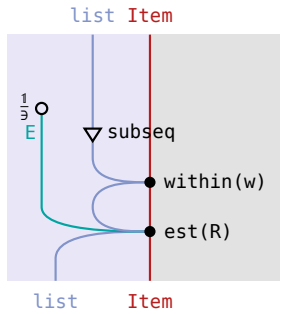
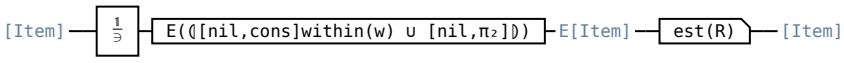
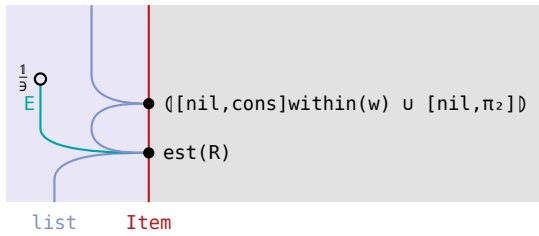
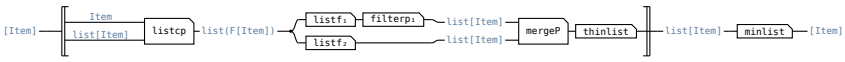
(8.4b)

the law	what it says
(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
(𝟙×Q) (cons (within w))⊑cons (within w)Q (𝟙×Q)π₂⊑π₂Q ,	both halves are monotonic on Q once ties in value are broken by weight

(8.4c)

$$0 \le w \wedge (\forall i.\ 0 \le wt(i)) \wedge (\forall X.\ \forall Y.\ X \sqsubseteq Y \Longrightarrow sortF(X) \sqsubseteq sortF(Y)) \wedge$$
$$sortF([nil,cons]R[nil,cons]^\circ)listf_1 \sqsubseteq P([nil,cons])sortP \wedge$$
$$sortF([nil,\pi_2]R[nil,\pi_2]^\circ)listf_2 \sqsubseteq P([nil,\pi_2])sortP \wedge sortP\ filterp_1 \sqsubseteq E(within(w))sortP \wedge$$
$$F(sortP)listcp \sqsubseteq cp(F,[Item])sortF(F(R)) \wedge (sortP \times sortP)mergeP \sqsubseteq cup\ sortP \wedge sortP$$
$$thinlist \sqsubseteq thin(Q(vol,wt))sortP \wedge sortP\ minlist \sqsubseteq est(R) \Longrightarrow (\!|listcp(listf_1\ filterp_1, listf_2)mergeP$$
$$thinlist|\!)minlist \sqsubseteq \frac{subseq\ within(w)}{\ni}\ est(R)$$

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)	
=  $\frac{(\!\|[nil,cons](within\ w) \cup [nil,\pi_2]\|\!)}{\ni}$ est(R) fusion, weights non-negative.	
⊒  ⦇listcp ⟨g₁,g₂⟩ merge R thinlist(Q)⦈ minlist(R) binary thinning theorem, at P≜R, F linear, Q from (8.4b)	
= ⦇[nil,cpr ⟨h₁,h₂⟩ merge R thinlist(Q)]⦈ minlist(R) listcp=wrap+cpr, gᵢ=[list(nil),hᵢ] — ≜ within; minlist(R) is head, packings coming out in descending value	

§8.5. The paragraph problem

(8.5a)

Line=list⁺ Word, Para=list⁺ Line, FA=Word+Word×A, listcp=wrap+cpr.

new(a,xs)=[[a]]#xs, glue(a,xs)=[[a]#head(xs)]#tail(xs), partition≜⦇[wrap wrap,new ∪ glue]⦈ : list⁺ Word→Para.

width≜⦇[length,(length×𝟙) plus succ]⦈, 0≤length a, fits w the coreflexive on a line x with width x≤w, ok w the coreflexive on [x]#xs with width x≤w.

white w x=w−width x, collect≜list(sqr) sum, waste w≜init list(white w) collect.

R≜(waste w)≤(waste w)° , Q≜R∩(head head°) , P≜⊤ , .

g₁≜list([wrap wrap,new]), g₂≜list([wrap wrap,glue]) filter(ok w), start≜wrap wrap wrap, h₁≜list(new), h₂≜list(glue) filter(ok w).

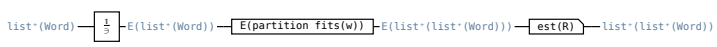
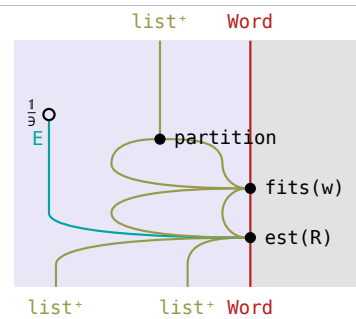
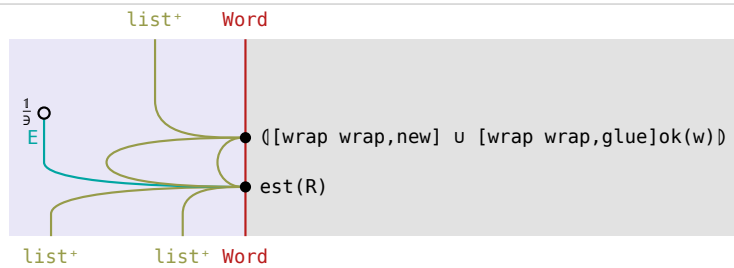
(8.5b)

the law	what it says
(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
(𝟙×Q) new⊑new Q (𝟙×Q) (glue (ok w))⊑glue (ok w)Q cons monotonic on collect≤collect° ,	both halves are monotonic on Q once ties in waste are broken by the first line
merge ⊤=cat; P≜head prefix head° also serves	⊤ needs no sorting at all, and prefix is a linear order on first lines of paragraphs of one input

(8.5c)

$$(\forall a.\ 0\le len(a)) \wedge (\forall a.\ len(a)\le w) \wedge (\forall X.\ \forall Y.\ X\sqsubseteq Y \Longrightarrow sortF(X)\sqsubseteq sortF(Y)) \wedge sortF([wrap\ wrap,new]\top[wrap\ wrap,new]^\circ)listf_1\sqsubseteq P([wrap\ wrap,new])sortP \wedge sortF([wrap\ wrap,glue]\top[wrap\ wrap,glue]^\circ)listf_2\sqsubseteq P([wrap\ wrap,glue])sortP \wedge sortP\ filterp_2\sqsubseteq E(ok(w))sortP \wedge F(sortP)listcp\sqsubseteq cp(F,list^+((list^+\ Word)))sortF(F(\top)) \wedge (sortP\times sortP)mergeP\sqsubseteq cup\ sortP \wedge sortP\ thinlist\sqsubseteq thin(Q(len,w))sortP \wedge sortP\ minlist\sqsubseteq est(R) \Longrightarrow ⦇listcp(listf_1,listf_2\ filterp_2)mergeP\ thinlist⦈minlist\sqsubseteq \frac{partition\ fits(w)}{\ni}\ est(R)$$

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

circuit	Hinze–Marsden
 partition list⁺(fits w) / ∋ est(R)	
= ⦇[wrap wrap,new ∪ (glue (ok w))]⦈ / ∋ est(R) fusion, every word fits on a line by itself.	
= ⦇[wrap wrap,new] ∪ ([wrap wrap,glue] (ok w))⦈ / ∋ est(R) the algebra as (f₁p₁) ∪ (f₂p₂), p₁≜𝟙 — ≜ thinlist.	
⊑ ⦇listcp ⟨g₁,g₂⟩ cat thinlist(Q)⦈ minlist(R) binary thinning theorem, at P≜⊤ with merge ⊤=cat, Q from (8.5b)	
= ⦇[start,cpr ⟨h₁,h₂⟩ cat thinlist(Q)]⦈ minlist(R) listcp=wrap+cpr, gᵢ along the coproduct — ≜ partition	

§8.6. Bitonic tours

(8.6a)

FA=(City×City)+(City×A), the base functor of cons-lists of length at least two; listcp=wrap+cpr; tc : City×City→Real, neither positive nor symmetric.

start (a,b)=([a,b],[a,b]), dropl (a,([b]⧺xs,ys))=([a]⧺xs,[a]⧺ys), dropr (a,(xs,[b]⧺ys))=([a]⧺xs,[a]⧺ys), tour≜⦇[start,dropl ∪ dropr]⦈.

cost (xs,ys)=outcost xs+incost ys, outcost $[a_0,\ldots,a_n]$=tc $(a_0,a_1)+\cdots+$tc (a_{n-1},a_n), incost $[a_0,\ldots,a_n]$=tc $(a_1,a_0)+\cdots+$tc (a_n,a_{n-1}).

next≜tail head, next2≜next×next, head2≜head×head, R≜cost≤cost° , Q≜R∩(next2 next2°)∩(head2 head2°), P≜⊤ , , g_1≜list([start,dropl]), g_2≜list([start,dropr]).

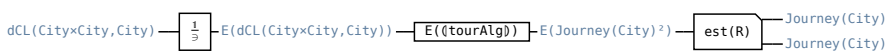
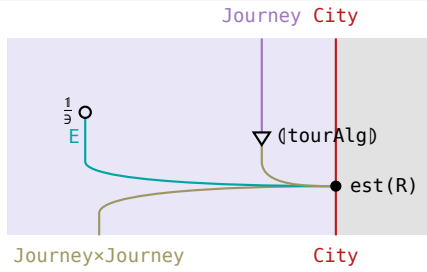
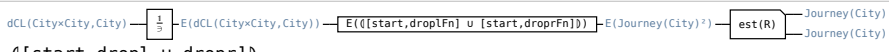
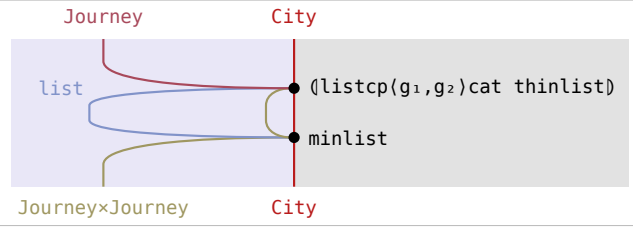
(8.6b)

the law	what it says
(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
(𝟙×Q) dropl⊑dropl Q (𝟙×Q) dropr⊑dropr Q ,	both are, once ties in cost are broken by the two second cities — the heads already agree among tours of one input

(8.6c)

$$(\forall X.\ \forall Y.\ X\sqsubseteq Y \Longrightarrow \mathrm{sortF}(X)\sqsubseteq\mathrm{sortF}(Y)) \wedge$$
$$\mathrm{sortF}([\mathrm{start},\mathrm{droplFn}]\top[\mathrm{start},\mathrm{droplFn}]^\circ)g_1\sqsubseteq P([\mathrm{start},\mathrm{droplFn}])\mathrm{sortP} \wedge$$
$$\mathrm{sortF}([\mathrm{start},\mathrm{droprFn}]\top[\mathrm{start},\mathrm{droprFn}]^\circ)g_2\sqsubseteq P([\mathrm{start},\mathrm{droprFn}])\mathrm{sortP} \wedge$$
$$F(\mathrm{sortP})\mathrm{listcp}\sqsubseteq\mathrm{cp}(F,\mathrm{dTour}(\mathrm{City}))\mathrm{sortF}(F(\top)) \wedge (\mathrm{sortP}\times\mathrm{sortP})\mathrm{cat}\sqsubseteq\mathrm{cup}\ \mathrm{sortP} \wedge \mathrm{sortP}$$
$$\mathrm{thinlist}\sqsubseteq\mathrm{thin}(Qc(\mathrm{tc}))\mathrm{sortP} \wedge \mathrm{sortP}\ \mathrm{minlist}\sqsubseteq\mathrm{est}(R) \Longrightarrow (\!|\mathrm{listcp}(g_1,g_2)\mathrm{cat}\ \mathrm{thinlist}|\!)\mathrm{minlist}\sqsubseteq \frac{(\!|\mathrm{tourAlg}|\!)}{\ni}\ \mathrm{est}(R)$$

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

circuit	Hinze–Marsden
 $\frac{\mathrm{tour}}{\ni}$ est(R)	
=  $\frac{(\!\|\mathrm{start},\mathrm{dropl}\ \cup\ \mathrm{dropr}\|\!)}{\ni}$ est(R) tour≜⦇[start,dropl ∪ dropr]⦈ — ≜ tour	
⊑ ⦇listcp ⟨g_1,g_2⟩ cat thinlist(Q)⦈ minlist(R) binary thinning theorem, at P≜⊤ with merge ⊤=cat, Q from (8.6b)	
= ⦇[start wrap,cpr ⟨list(dropl),list(dropr)⟩ cat thinlist(Q)]⦈ minlist(R) listcp=wrap+cpr, g_i=[list(start),list(drop_i)] — ≜ tour; quadratic, two tours added per step	

§9. Dynamic Programming

§9.1. Example of Hylo ⦇T⦈°⦇h⦈: segmenting a list

(9.1a)

- ⦇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

(9.1b)

x	h(x)
inl(())	[]
inr(([c],[]))	[c]
inr(([a, b],[c]))	[a + b, c]
inr(([a],[b + c]))	[a, b + c]

(9.1c)

- 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

§9.2. Theory

(9.2a)

h : FB→B a map, T : FA→A an F-algebra, R : B→B.

H≜⦇T⦈°⦇h⦈ : A→B, M≜ $\frac{H}{\ni}$ est(R) the problem to be solved, (μX : G(X)) as in ≜ μ.

DPSetting(h,R)≜Map(h) ∧ MonoAlg(h,R°) ∧ R°R°⊑R°

h is a map, h is monotonic on R°, and R° is transitive — assumed by every theorem below

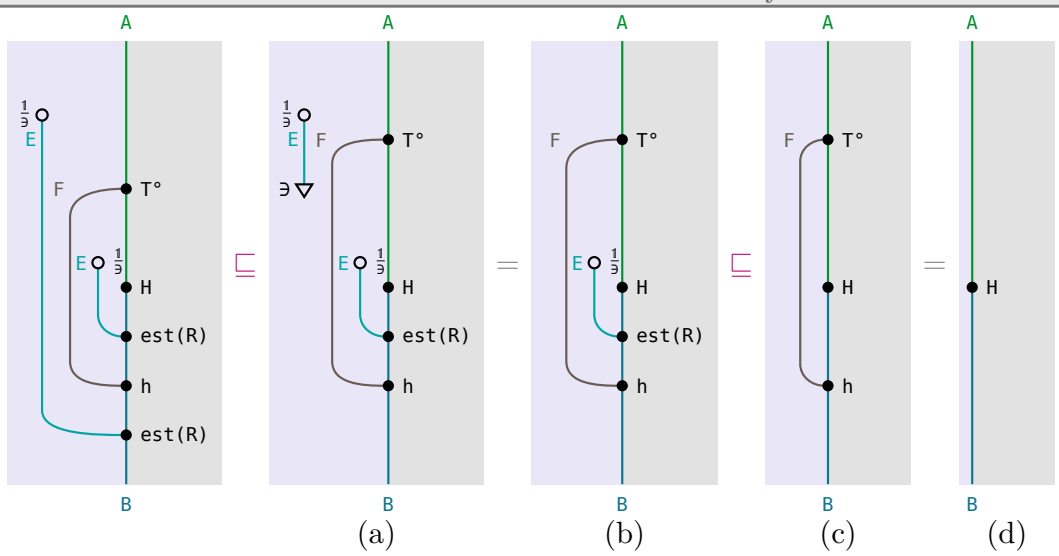
(μX : $\frac{T^\circ}{\ni}$ P(F(X)h)est(R))⊑$\frac{H}{\ni}$ est(R) (9.2b)

every answer the recursion returns is an optimal one

- $\frac{T^\circ}{\ni}$ 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

$$\mathtt{T^\circ F(H)h=H} \implies \tfrac{\mathtt{T^\circ}}{\ni}\ \mathtt{P(F(}\tfrac{\mathtt{H}}{\ni}\ \mathtt{est(R))h)est(R)}\sqsubseteq\mathtt{H} \tag{9.2c}$$

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



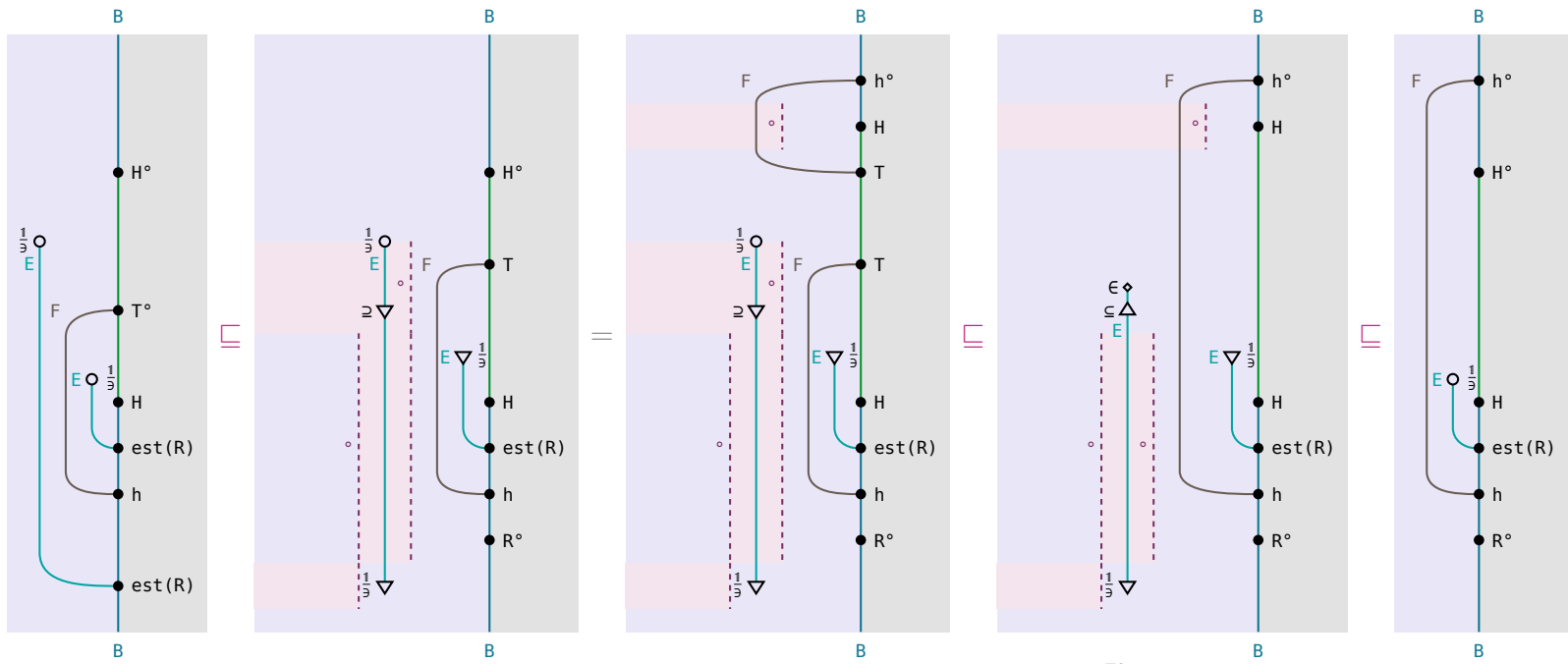
(a) $\mathtt{P(X)est(R)}\sqsubseteq\ni\mathtt{X}$ — (7.1.8a)
(b) $\frac{\mathtt{T^\circ}}{\ni}\ni=\mathtt{T^\circ}$ — (3.1j)
(c) $\mathtt{M}\sqsubseteq\frac{\mathtt{H}}{\ni}\ni=\mathtt{H}$ — (7.1.1a)
(d) $\mathtt{T^\circ F(H)h=H}$ — hylomorphism theorem

$$\mathtt{T^\circ F(H)h{=}H} \implies \mathtt{H^\circ}\tfrac{\mathtt{T^\circ}}{\ni}\ \mathtt{P(F(}\tfrac{\mathtt{H}}{\ni}\ \mathtt{est(R))h)est(R)}\sqsubseteq\mathtt{R^\circ} \tag{9.2d}$$

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

$$\mathtt{T^\circ F(H)h{=}H} \implies \mathtt{H^\circ}\tfrac{\mathtt{T^\circ}}{\ni}\ \mathtt{P(F(}\tfrac{\mathtt{H}}{\ni}\ \mathtt{est(R))h)est(R)}\sqsubseteq\mathtt{h^\circ F(H^\circ)F(}\tfrac{\mathtt{H}}{\ni}\ \mathtt{est(R))hR^\circ}$$

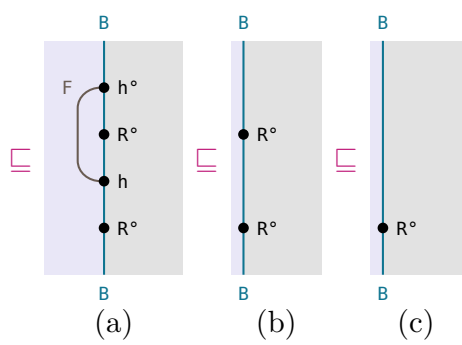
from an answer of H, the step's answer is reached by taking H's parts back, rebuilding them by F(M)h, then one R°



(a) P(X)est(R)⊑∈\(XR°) — (7.1.8a)
(b) H°=h°F(H°)T — hylomorphism theorem

$$\mathtt{h^\circ F(H^\circ)F(}\tfrac{\mathtt{H}}{\ni}\ \mathtt{est(R))hR^\circ}\sqsubseteq\mathtt{R^\circ}$$

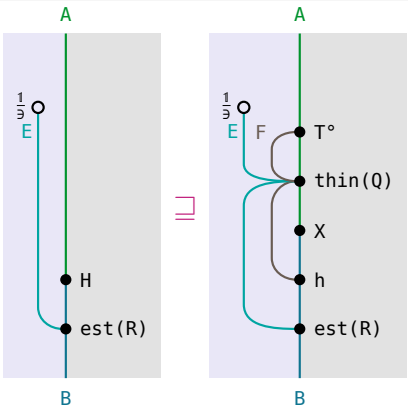
rebuilding an answer of H from its parts, each replaced by M's, and one R° stays within R°



(a) H°M⊑R° — (7.1.1a)
(b) h monotonic on R°
(c) R transitive

ThinCondition(T,h,R,Q) ⟹ (μX : $\frac{T^\circ}{\ni}$ thin(Q)P(F(X)h)est(R))⊑$\frac{H}{\ni}$ est(R) (9.2e)

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



(a) (b)

(a) H≜⦇T⦈°⦇h⦈ — ≜ H
(b) QF(H)h⊑F(H)hR, h monotonic on R — (8.1b), (7.1.1a)

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

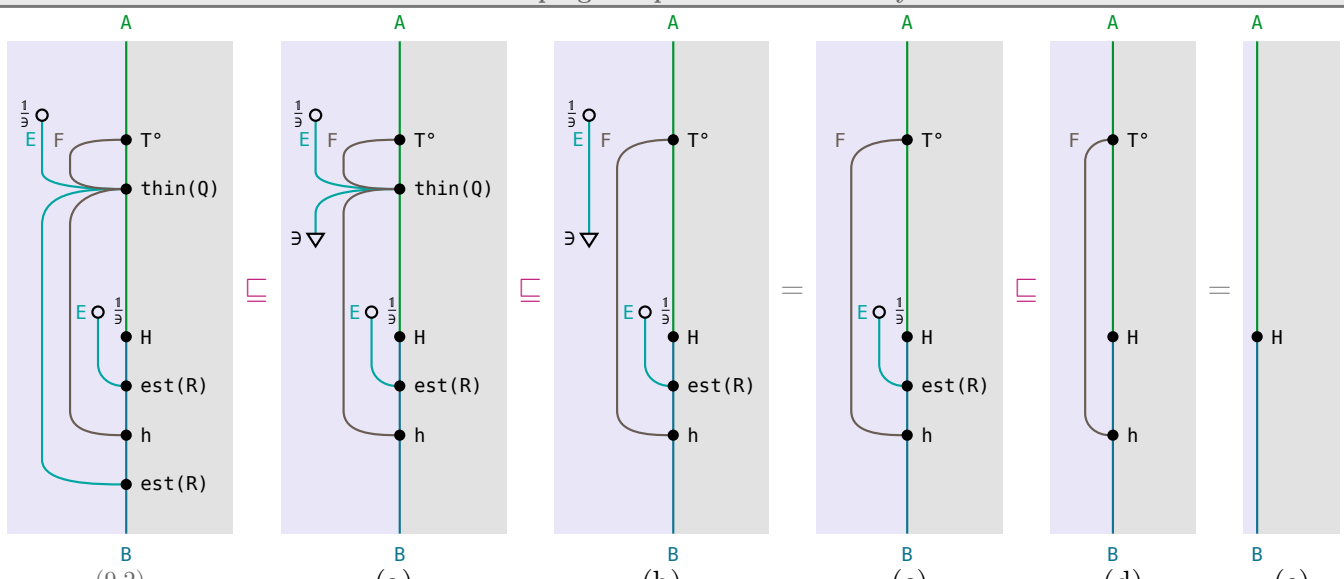
QF(H)h⊑F(H)hR: every result of thinning by Q, solving the parts by F(H) and assembling by h is reached by R from a result of solving by F(H) and assembling by h without Q

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

T°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

T°F(H)h=H ⟹ $\frac{T^\circ}{\ni}$ thin(Q)P(F($\frac{H}{\ni}$ est(R))h)est(R)⊑H (9.2f)

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



(9.2) (a) (b) (c) (d) (e)

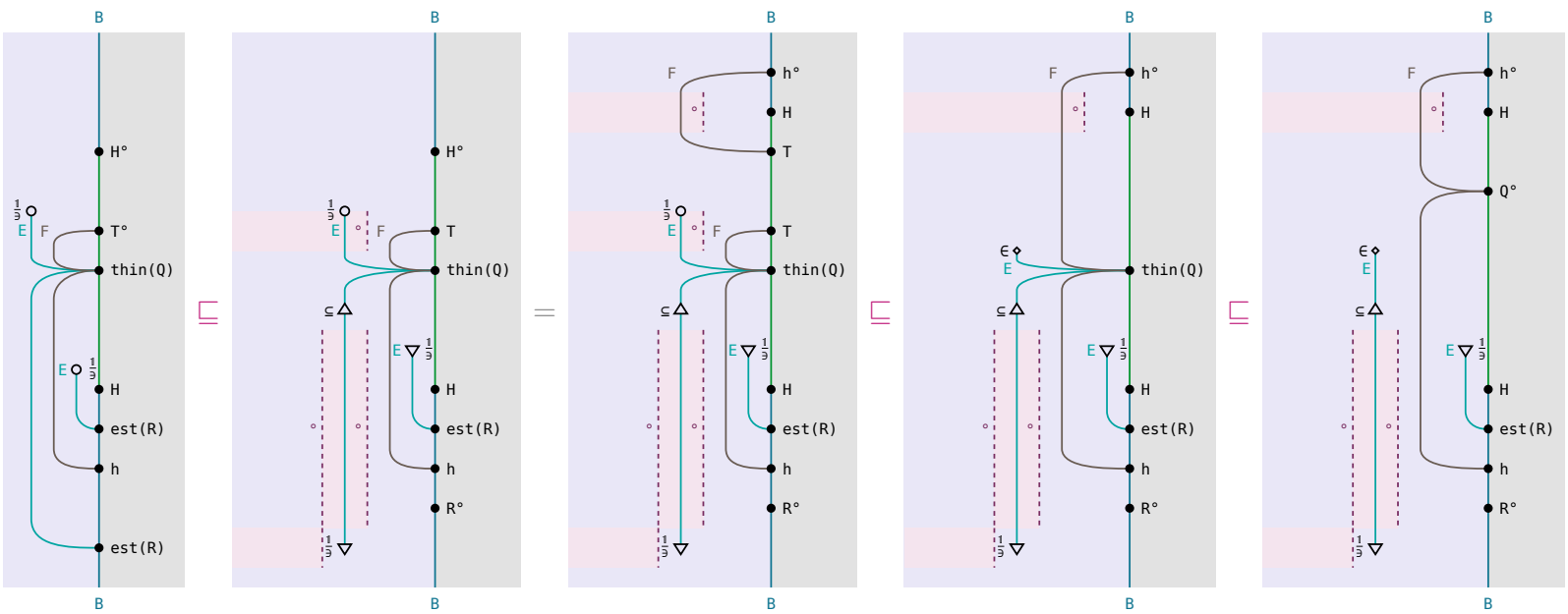
(a) P(X)est(R)⊑∋X — (7.1.8a)
(b) thin(Q)∋⊑∋ — (8.1b)
(c) $\frac{T^\circ}{\ni}$∋=T° — (3.1j)
(d) M⊑$\frac{H}{\ni}$∋=H — (7.1.1a)
(e) T°F(H)h=H — hylomorphism theorem

$$\mathtt{T^\circ F(H)h{=}H} \wedge \mathtt{QF(H)h} \sqsubseteq \mathtt{F(H)hR} \implies \mathtt{H^\circ}\tfrac{\mathtt{T^\circ}}{\ni}\ \mathtt{thin(Q)P(F(}\tfrac{\mathtt{H}}{\ni}\ \mathtt{est(R))h)est(R)} \sqsubseteq \mathtt{R^\circ} \tag{9.2g}$$

$\mathtt{H^\circ}$ followed by the body at $\mathtt{M}$ is $\sqsubseteq\mathtt{R^\circ}$: an answer of the body is never worse than an answer of $\mathtt{H}$ to the same input

$$\mathtt{T^\circ F(H)h{=}H} \implies \mathtt{H^\circ}\tfrac{\mathtt{T^\circ}}{\ni}\ \mathtt{thin(Q)P(F(}\tfrac{\mathtt{H}}{\ni}\ \mathtt{est(R))h)est(R)} \sqsubseteq \mathtt{h^\circ F(H^\circ)Q^\circ}\in(\in\backslash\mathtt{F(}\tfrac{\mathtt{H}}{\ni}\ \mathtt{est(R))hR^\circ})$$

from an answer of $\mathtt{H}$, the thinned step's answer is reached through $\mathtt{H}$'s parts, a $\mathtt{Q^\circ}$ step to a candidate, and that candidate's answer

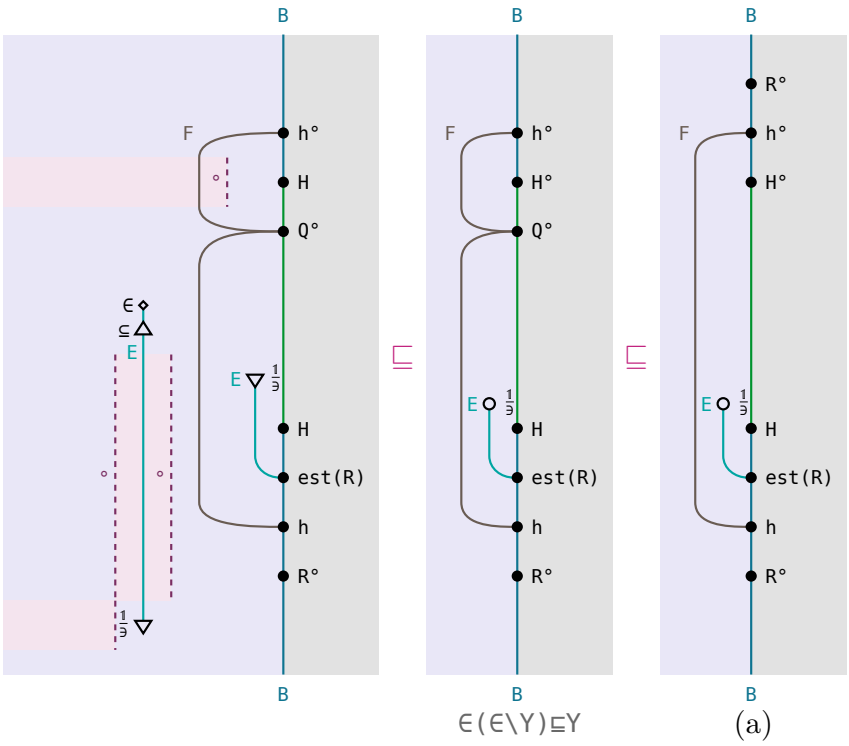


(a) $\mathtt{P(X)est(R)} \sqsubseteq \in\backslash\mathtt{(XR^\circ)}$ — (7.1.8a)
(b) $\mathtt{H^\circ{=}h^\circ F(H^\circ)T}$ — hylomorphism theorem

(9.2h)

$$\mathsf{QF(H)h \sqsubseteq F(H)hR} \implies \mathsf{h^\circ F(H^\circ)Q^\circ \in (\in\backslash F(\tfrac{H}{\ni}\ est(R))hR^\circ) \sqsubseteq R^\circ h^\circ F(H^\circ)F(\tfrac{H}{\ni}\ est(R))hR^\circ}$$

the hypothesis on Q turns the Q° step into an R° step in front of rebuilding by F(M)h

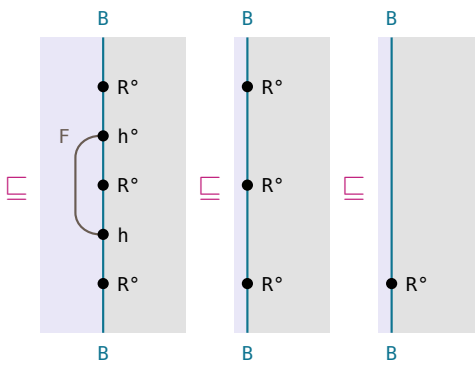


∈(∈\Y)⊑Y (a)

(a) QF(H)h⊑F(H)hR conversed

$$\mathsf{R^\circ h^\circ F(H^\circ)F(\tfrac{H}{\ni}\ est(R))hR^\circ \sqsubseteq R^\circ}$$

rebuilding from parts replaced by M's, between two R° steps, stays within R°

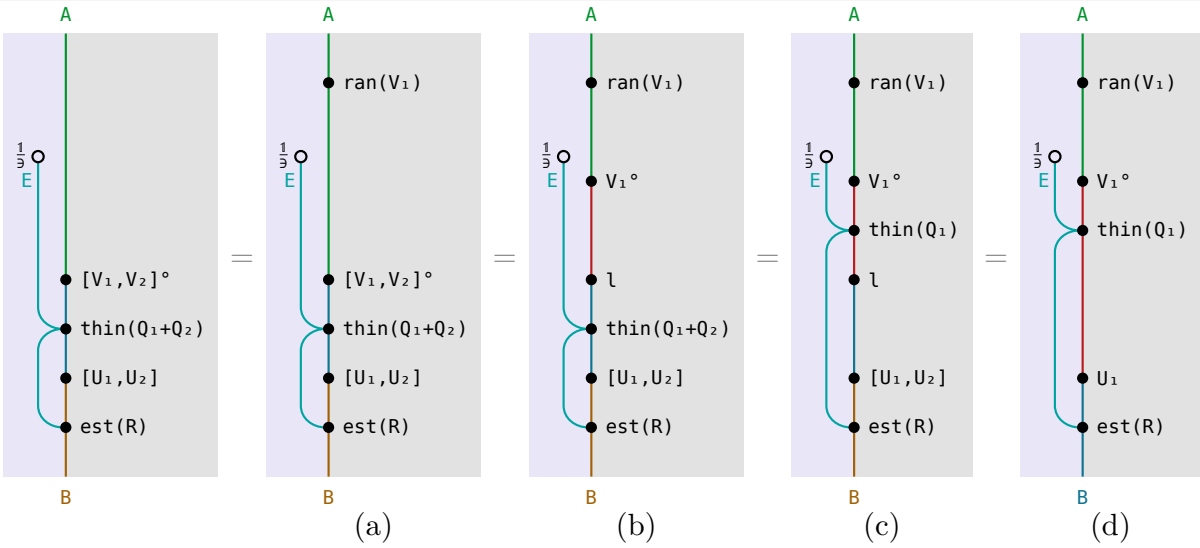


(a) (b) (c)

(a) H°M⊑R° under F — (7.1.1a)
(b) h°F(R°)h⊑R°
(c) R transitive, twice

$$V_2\ V_1{}^\circ = \mathbb{0} \implies \frac{[V_1,V_2]^\circ}{\ni}\ \mathrm{thin}(Q_1+Q_2)P([U_1,U_2])\mathrm{est}(R) = \mathrm{ran}(V_1)\frac{V_1{}^\circ}{\ni}\ \mathrm{thin}(Q_1)P(U_1)\mathrm{est}(R) \cup \mathrm{ran}(V_2)\frac{V_2{}^\circ}{\ni}\ \mathrm{thin}(Q_2)P(U_2)\mathrm{est}(R) \tag{9.2i}$$

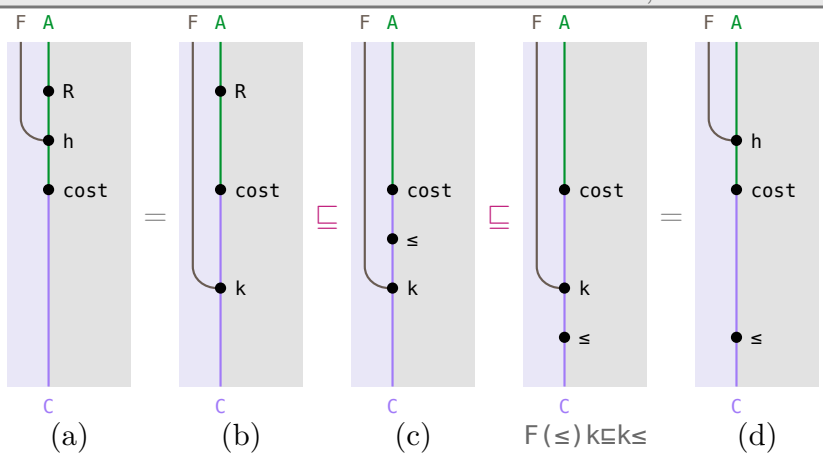
when V_1 and V_2 have disjoint ranges, the body over $[V_1,V_2]$ runs the V_1 problem on inputs V_1 reaches and the V_2 problem on inputs V_2 reaches



(a) empty off $\mathrm{ran}\ V_1 \cup \mathrm{ran}\ V_2$
(b) $\frac{[V_1,V_2]^\circ}{\ni} = \frac{V_1{}^\circ}{\ni}P(\mathrm{inl})$ on $\mathrm{ran}\ V_1$
(c) $P(\mathrm{inl})\mathrm{thin}(Q_1+Q_2)=\mathrm{thin}(Q_1)P(\mathrm{inl})$
(d) P a relator, $\mathrm{inl}[U_1,U_2]=U_1$

$$\mathrm{Map}(\mathrm{cost}) \wedge R=\mathrm{cost}\leq\mathrm{cost}^\circ \wedge \mathrm{hcost}=F(\mathrm{cost})k \wedge F(\leq)k\sqsubseteq k\leq \implies \mathrm{MonoAlg}(h,R) \tag{9.2j}$$

if R compares two values by comparing their costs under $\leq$, and h then cost equals $F(\mathrm{cost})$ then a k monotonic on $\leq$, then h is monotonic on R



(a) definition of R, shunting
(b) $h\ \mathrm{cost}=F(\mathrm{cost})k$
(c) $R\ \mathrm{cost}\sqsubseteq\mathrm{cost}\ \leq$
(d) $h\ \mathrm{cost}=F(\mathrm{cost})k$

(9.2k)

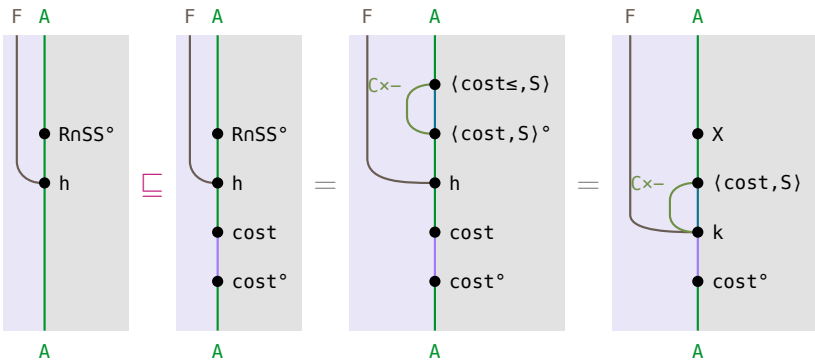
the condition	how it is discharged
$F(R)h\sqsubseteq hR$ Proposition 9.2, $R\triangleq\mathrm{cost}\leq\mathrm{cost}^\circ$, $h\ \mathrm{cost}=F(\mathrm{cost})k$, $F(\leq)k\sqsubseteq k\leq$	monotonicity when the cost is itself a fold with a step k monotonic on $\leq$
$F(R\cap(H^\circ H))h\sqsubseteq hR$ Proposition 9.3, $R\triangleq\mathrm{cost}\leq\mathrm{cost}^\circ$, $h\ \mathrm{cost}=F(\langle\mathrm{cost},H^\circ\rangle)k$, $F(\leq\times\mathbb{1})k\sqsubseteq k\leq$, H° simple;	monotonicity **in context**: k may also read the input the part was built from
$QF(H)h\sqsubseteq F(H)hR$ at $Q\triangleq F(U,V)$ Proposition 9.4, U, V preorders, $F(U,R)h\sqsubseteq hR$, $VH\sqsubseteq HR$	both conditions at once, split along the two arguments of a bifunctor

Map(cost) ∧ Simple(S) ∧ R=cost≤cost° ∧ hcost=F(⟨cost,S⟩)k ∧ F(≤×𝟙)k⊑k≤ ⟹ F(R∩SS°)h⊑hR (9.21)

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

Map(cost) ∧ R=cost≤cost° ∧ hcost=F(⟨cost,S⟩)k ⟹ F(R∩SS°)h⊑F(⟨cost≤,S⟩⟨cost,S⟩°)F(⟨cost,S⟩)kcost°

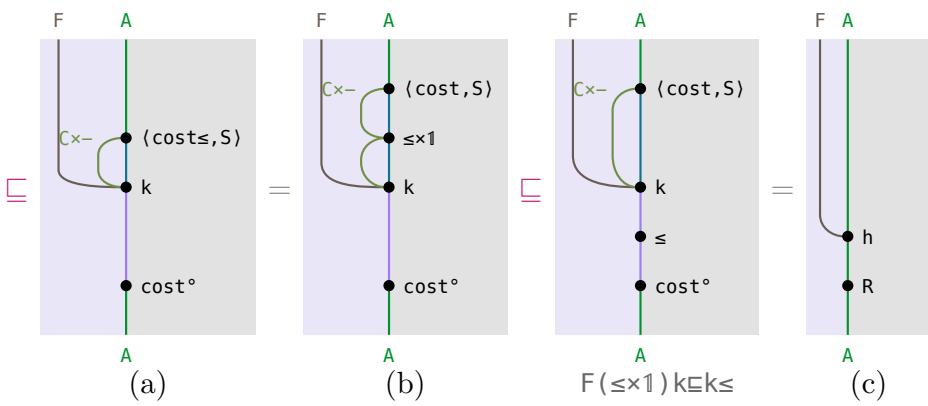
improving the parts within their S-context and assembling by h is below the same improvement on the (cost, context) bundles, assembled by k



(a) 𝟙⊑cost cost° — (3.1c)
(b) R∩SS°=⟨cost≤,S⟩⟨cost,S⟩° — ≜ ×
(c) h cost=F(⟨cost,S⟩)k

Map(cost) ∧ Simple(S) ∧ R=cost≤cost° ∧ hcost=F(⟨cost,S⟩)k ∧ F(≤×𝟙)k⊑k≤ ⟹ F(⟨cost≤,S⟩⟨cost,S⟩°)F(⟨cost,S⟩)kcost°⊑hR

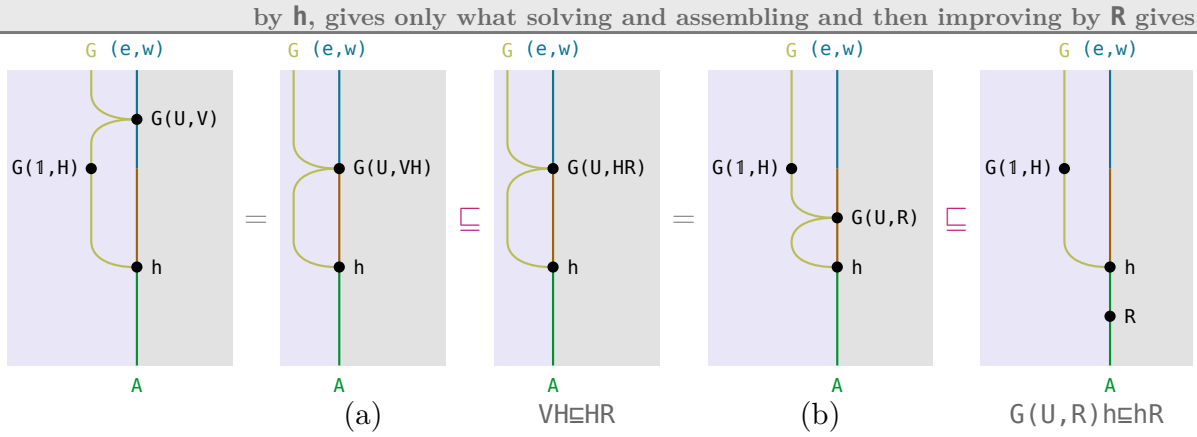
the improvement on the bundles, assembled by k, is below assembling by h and improving by R



(a) ⟨cost,S⟩°⟨cost,S⟩⊑𝟙
(b) ⟨cost≤,S⟩=⟨cost,S⟩(≤×𝟙) — (5.2.2b), (5.1b)
(c) h cost=F(⟨cost,S⟩)k, R=cost≤cost°

G(U,R)h⊑hR ∧ VH⊑HR ⟹ G(U,V)G(𝟙,H)h⊑G(𝟙,H)hR (9.2m)

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**)



(a) G(U,V)G(𝟙,H)=G(U,VH) — (5.1b)
(b) G(U,HR)=G(𝟙,H)G(U,R) — (5.1b)

§9.3. The string edit problem

(9.3a)

- `(∃a. o=cpy(a)) ∨ (∃a. o=del(a)) ∨ (∃a. o=ins(a))`
  an edit operation copies, deletes or inserts one character
- `FX=1+(Op×X)`
  `F(Op,X)` is the empty sequence or one operation paired with an `X`
- `edit=⦇[base,step]⦈`
  `edit` is the fold of `[base,step]` over an edit sequence, returning the two strings it reconstitutes
- `length=⦇[zero,π₂ succ]⦈`
  the fold of `[zero,π₂ succ]` counts the operations of a sequence
- `R=length≤length°`
  `R` relates `es` to `fs` when `es` is no longer than `fs`
- `V=suffix°×suffix°`
  `V` relates two pairs of strings when each string of the first is a suffix of the matching string of the second
- `Q(inl(u),inl(u))`
  `Q` is the identity on the `base` summand
- `Q(inr(p),inr(q)) ⟺ V(π₂(p),π₂(q))`
  on the `step` summand `Q` compares any two operations (`U≜T`) and the two pairs of strings by `V`
- `empty(p,q) ⟺ p=q ∧ p=([],[])`
  `empty` is the coreflexive holding only at the pair of two empty strings
- `q ∈ unstep(p) ⟹ step(inr(q))=p`
  every decomposition `unstep` returns is sent back to its input by `step`

(9.3b)

x	α(x)
inl(u)	wrap(u)
inr((o,es))	[o]⧺es

(9.3c)

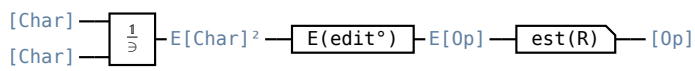
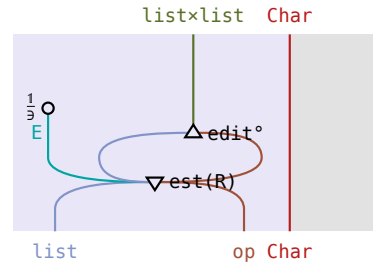
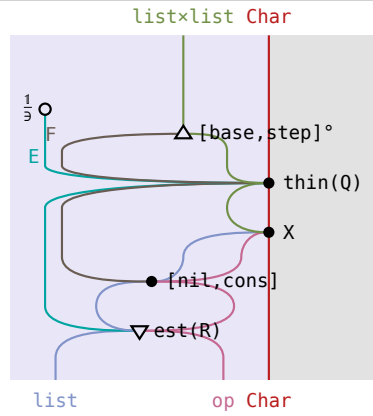
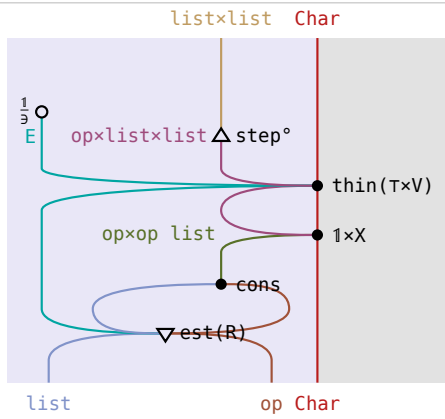
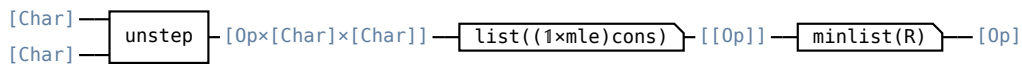
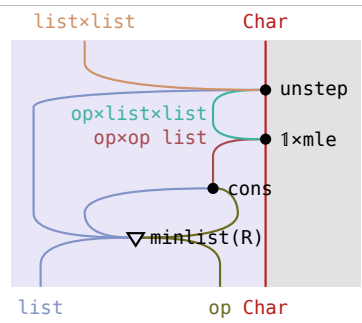
x	[base,step](x)
inl(u)	([],[])
inr((cpy(a),(xs,ys)))	([a]⧺xs,[a]⧺ys)
inr((del(a),(xs,ys)))	([a]⧺xs,ys)
inr((ins(a),(xs,ys)))	(xs,[a]⧺ys)

(9.3d)

p	unstep(p)
([],[])	[]
([a]⧺xs,[])	[(del(a),(xs,[]))]
([],[b]⧺ys)	[(ins(b),([],ys))]
([a]⧺xs,[b]⧺ys)	if a = b then [(cpy a, xs, ys)] else [(del a, xs, [b]⧺ys), (ins b, [a]⧺xs, ys)]

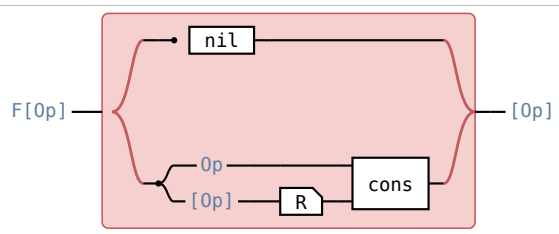
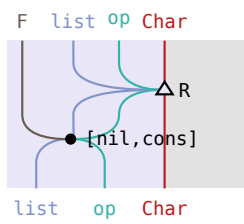
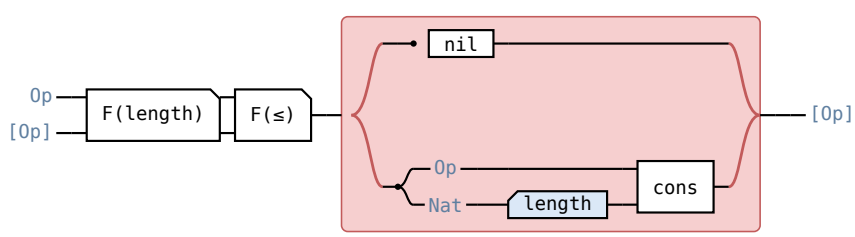
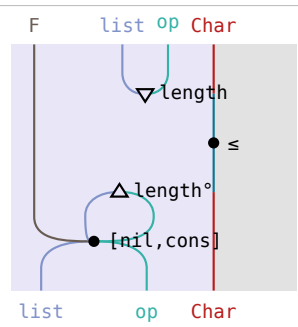
(μX : $\frac{\texttt{[base,step]°}}{\ni}$ thin(Q)P(F(X)[nil,cons])est(R))⊑$\frac{\texttt{edit°}}{\ni}$ est(R) , mle=(empty→nil,unstep list((𝟙×mle)cons) minlist(R)) (9.3e)

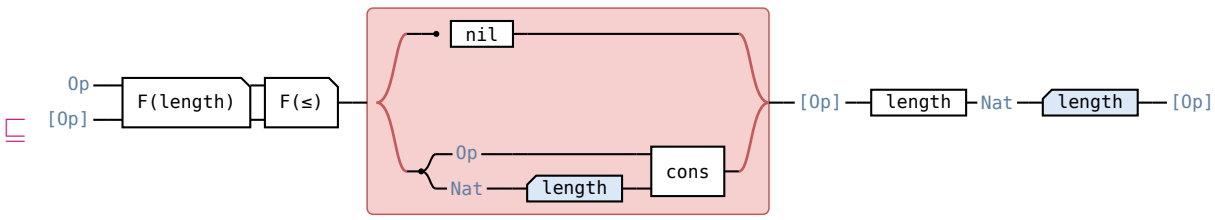
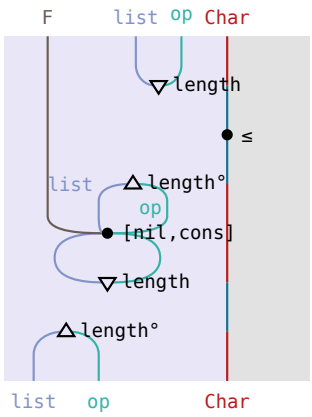
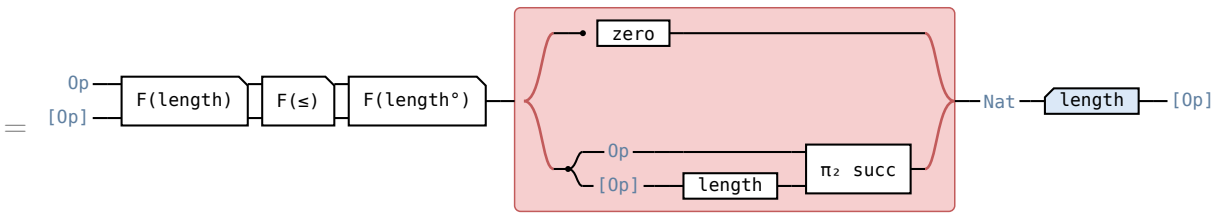
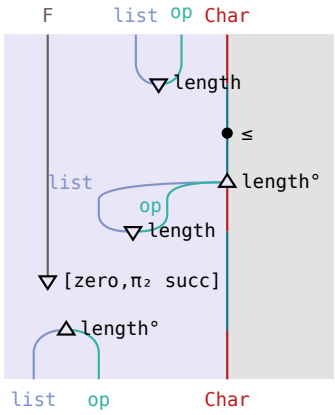
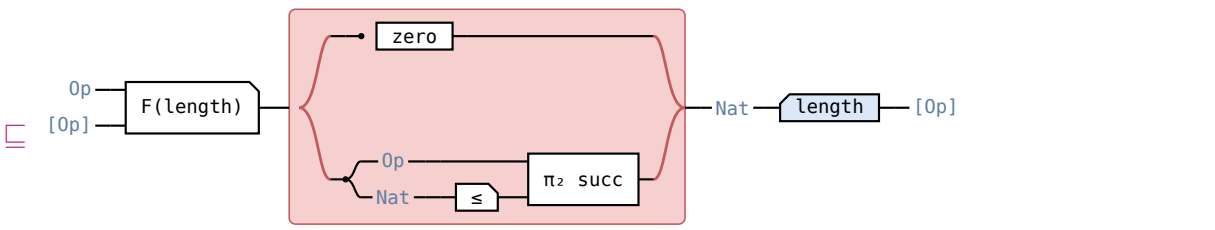
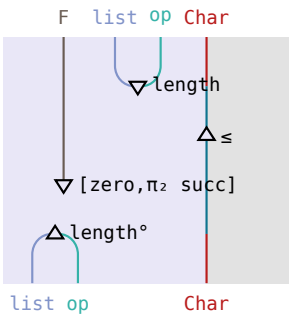
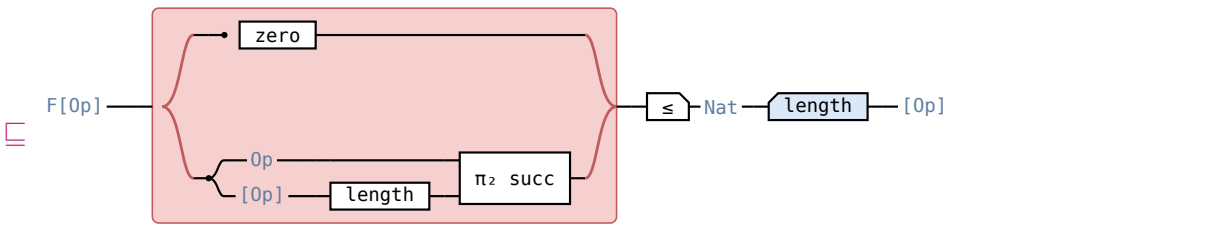
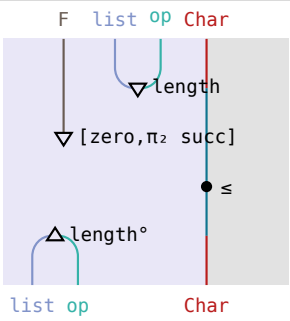
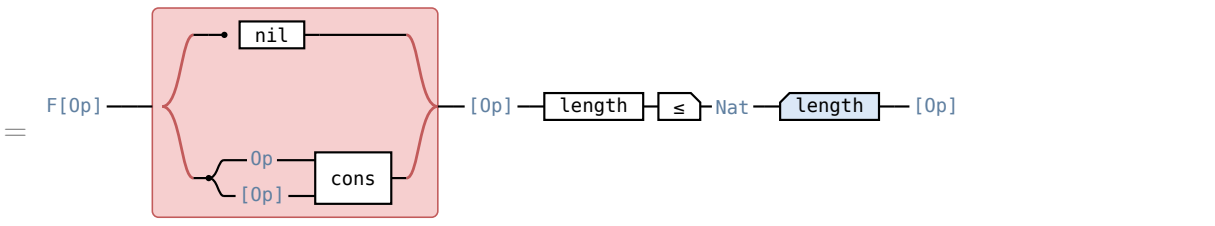
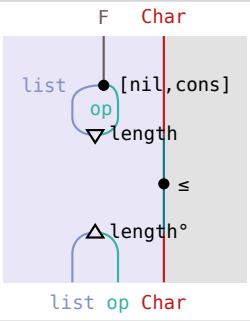
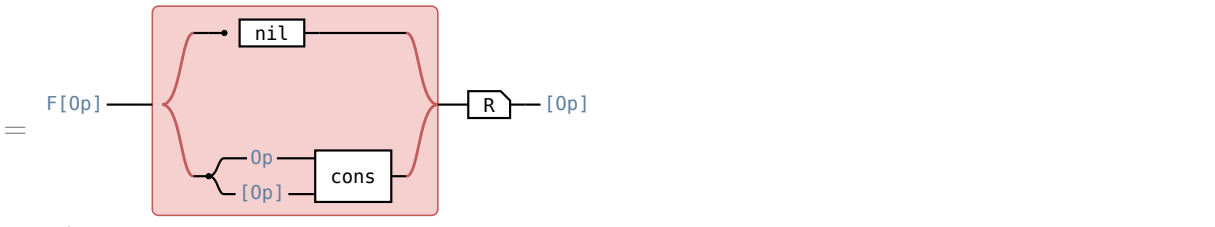
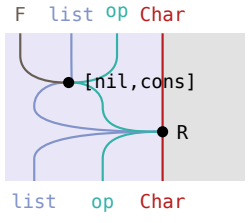
a shortest edit sequence from which both strings can be reconstituted is one pass over the two of them, each step copying, deleting or inserting one character and the best sequence for what is left taken from the entries already computed

circuit	Hinze–Marsden
 the specification — ≜ edit	
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 (9.3f)

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)

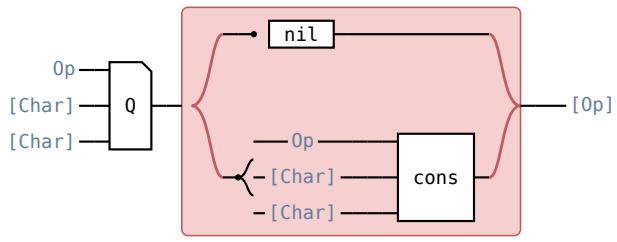
circuit	Hinze–Marsden
	
=  R≜length≤length° — ≜ edit; F preserves composition	

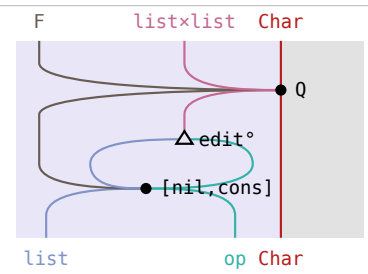
circuit	Hinze–Marsden
 length is a map, so entire: 𝟙⊑length length°	
 α length=F(length)[zero,π₂ succ]: length≜⦇[zero,π₂ succ]⦈ — ≜ edit	
 length is a map, so simple, and F is monotonic: F(length°)F(length)⊑F(𝟙)=𝟙	
 succ is monotonic on ≤: F(≤)[zero,π₂ succ]⊑[zero,π₂ succ]≤	
 F(length)[zero,π₂ succ]=α length again	
 R≜length≤length° — ≜ edit	

QF(edit°)[nil,cons]⊑F(edit°)[nil,cons]R (9.3g)

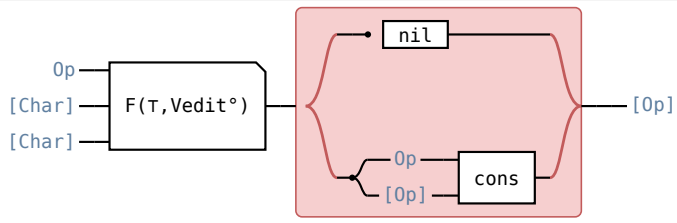
every edit sequence built from a decomposition Q puts above a given one — any operation, each string lengthened at the front — is at least as long as one built from the given one

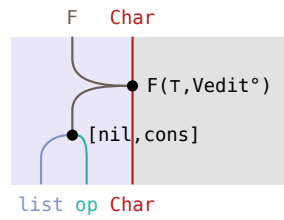
circuit	**Hinze–Marsden**





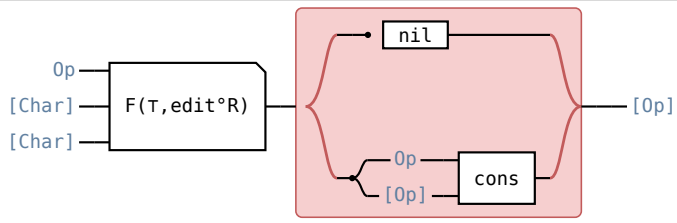
=

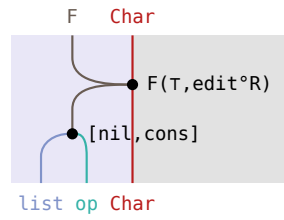




Q≜𝟙+(U×V) is F(U,V) at U≜T — ≜ edit; and the bifunctor F preserves composition: F(U,V)F(𝟙,edit°)=F(U,V edit°)

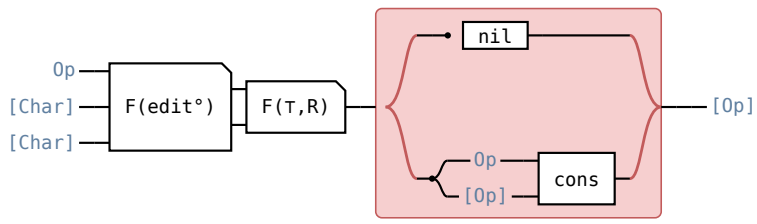
⊑

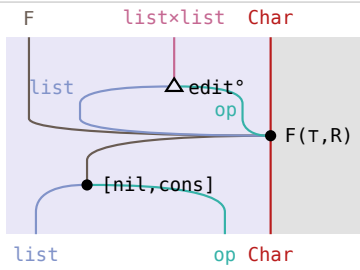




Proposition 9.4's second condition V edit°⊑edit° R — (9.3h)

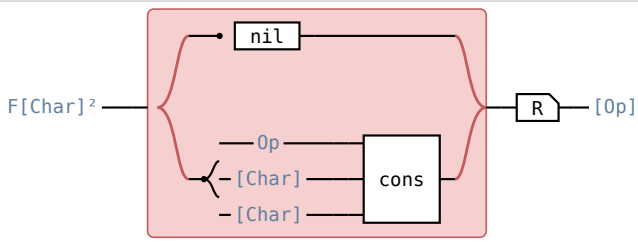
=

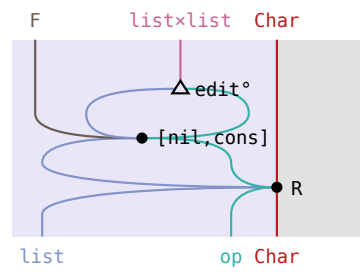




F preserves composition: F(U,edit° R)=F(𝟙,edit°)F(U,R)

⊑

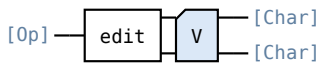
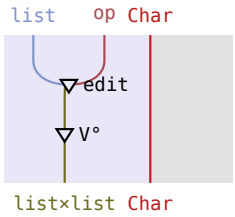
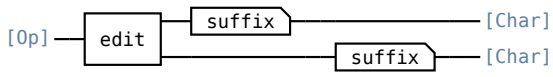
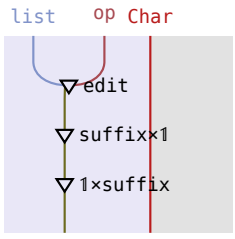
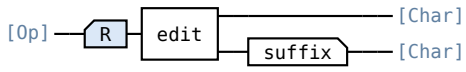
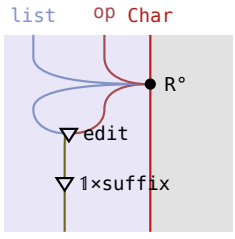
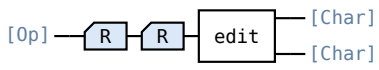
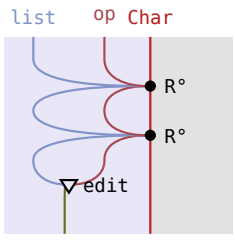
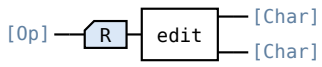
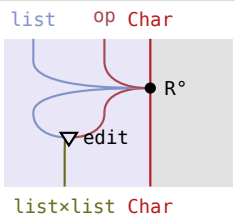




Proposition 9.4's first condition F(T,R)α⊑αR, left as an exercise in the book: cons adds one to both lengths whatever the two operations are

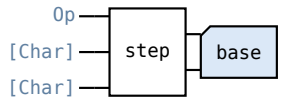
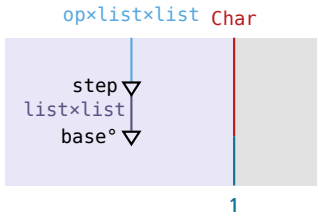
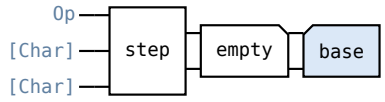
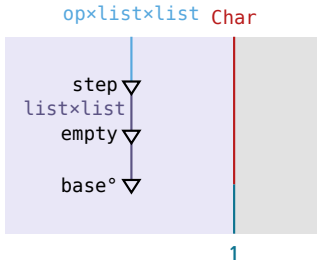
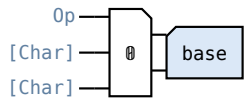
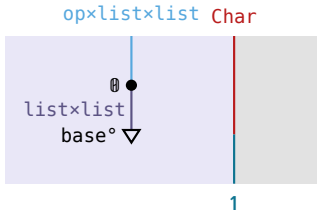
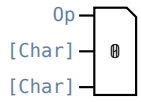
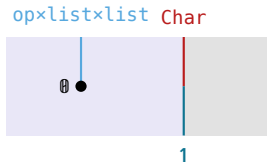
edit V°⊑R°edit (9.3h)

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 (9.3i)

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	

(9.3j)

= 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

= 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

= 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

§9.4. Optimal bracketing

(9.4a)

tree A::=tip A|bin (tree A,tree A), FX=A+X², so F(R)=𝟙+R²; h≜[tip,bin], flatten≜⦇[wrap,cat]⦈ : tree A→list⁺ A , H=flatten°.

⟨cost,size⟩≜⦇[opt,opb]⦈, opt≜⟨zero,st⟩, opb ((cx,sx),(cy,sy))=(cb (sx,sy)+cx+cy,sb (sx,sy)).

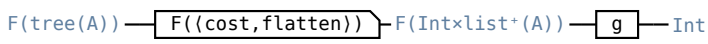
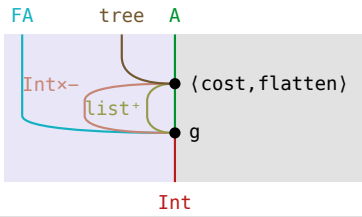
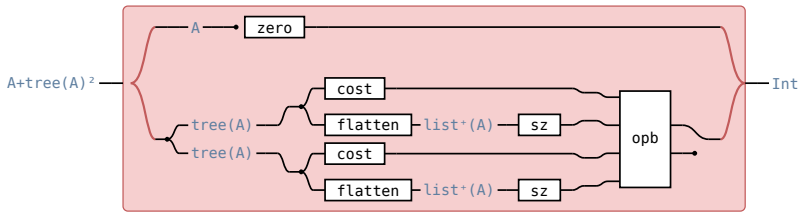
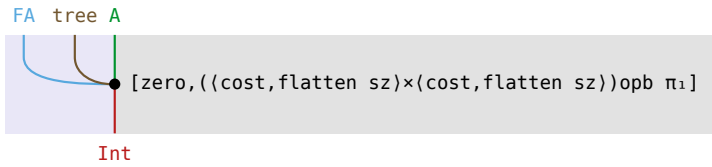
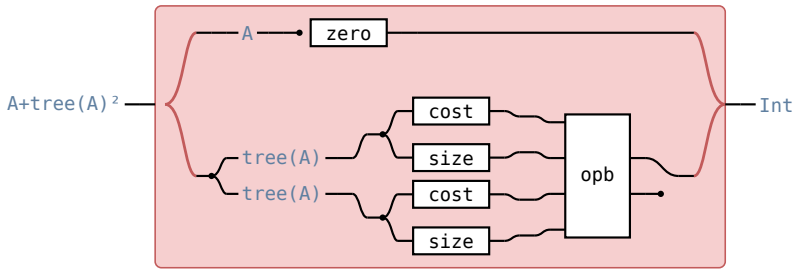
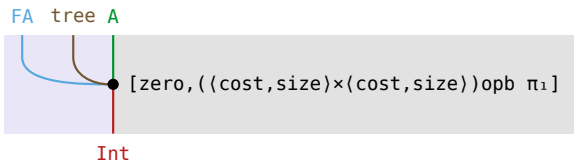
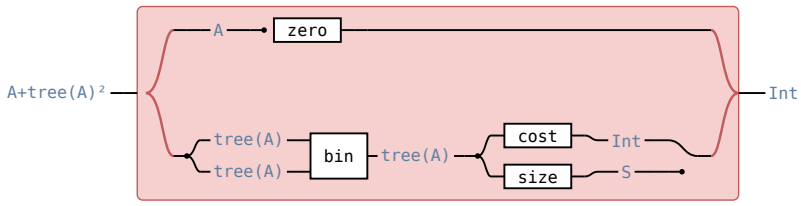
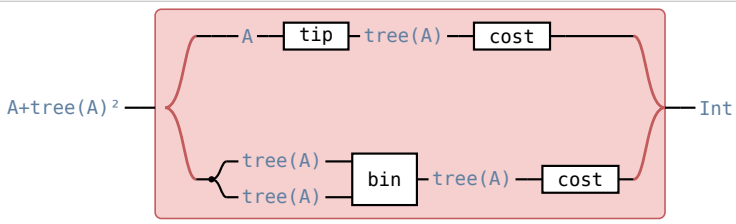
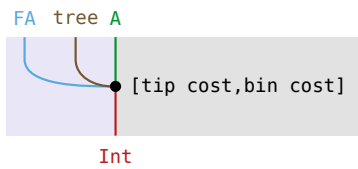
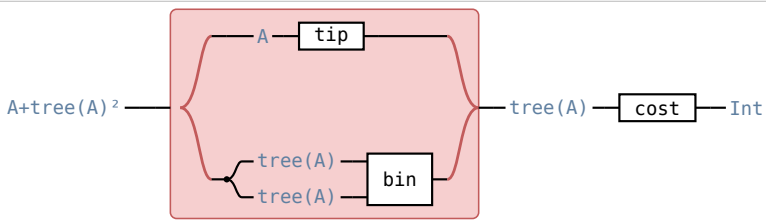
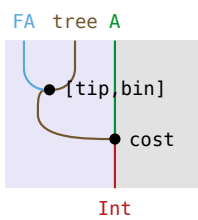
sb associative, so size=flatten sz for a map sz R≜cost≤cost° , g≜[zero,(𝟙×sz)² opb π₁], single the coreflexive on singleton lists.

splits≜⟨inits⁺,tails⁺⟩ zip, an implementation of $\frac{\text{cat}^\circ}{\ni}$; array≜inits list(row), row≜tails list(mct), col≜inits list(mct).

mix≜zip list(bin) minlist(R), next≜⟨π₁,mix⟩ snoc, process≜((tip wrap)×𝟙) loop(next).

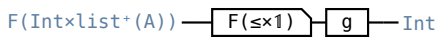
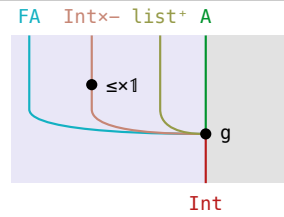
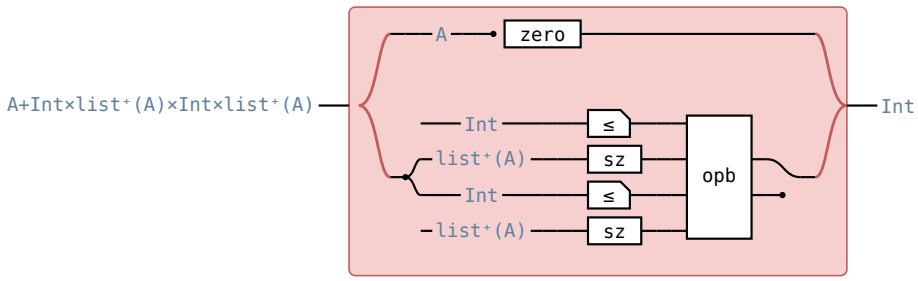
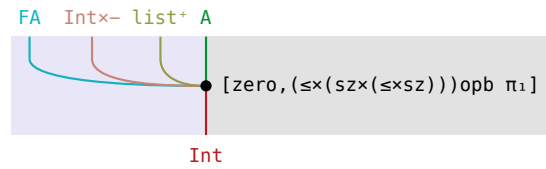
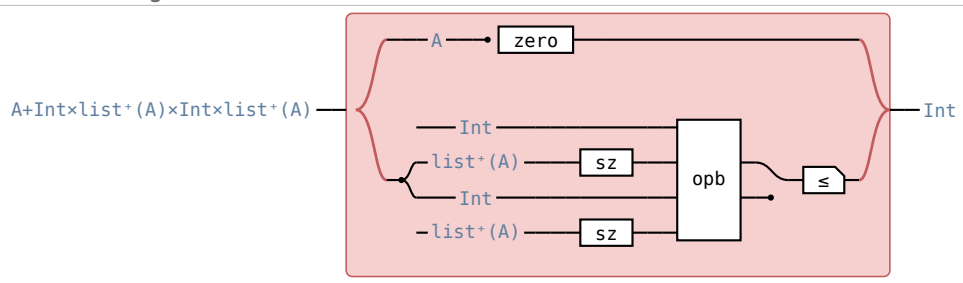
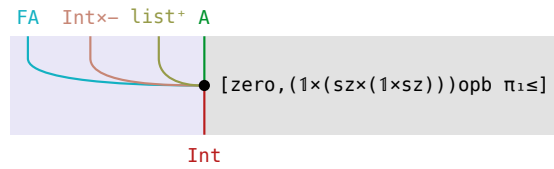
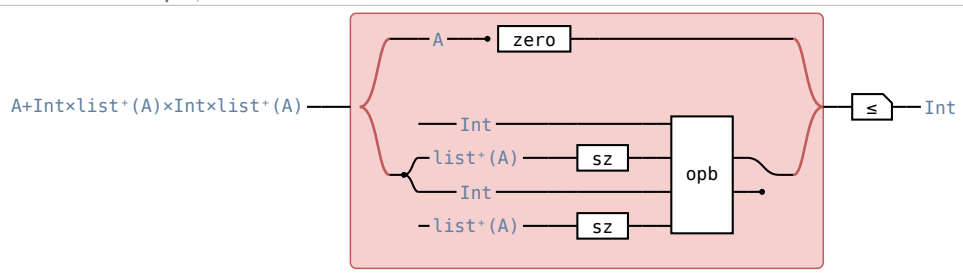
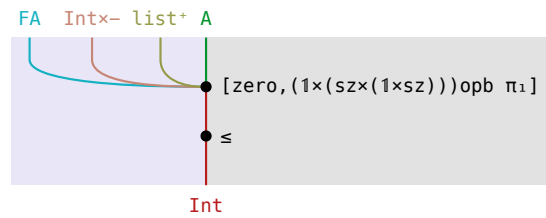
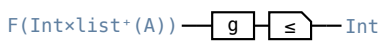
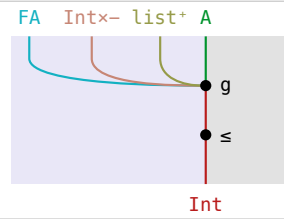
Assoc(sb) ⟹ F((cost,flatten))g=[tip,bin]cost (9.4b)

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 — ≜ flatten	
=  flatten sz=size, since sb is associative — ≜ flatten	
=  ⟨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	

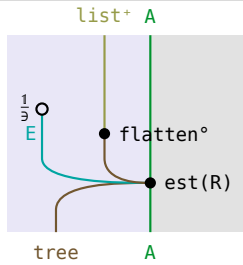
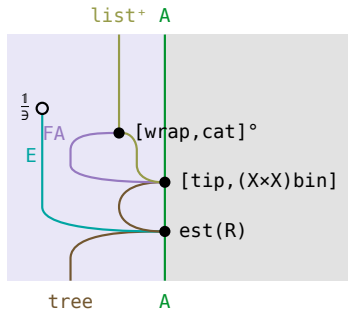
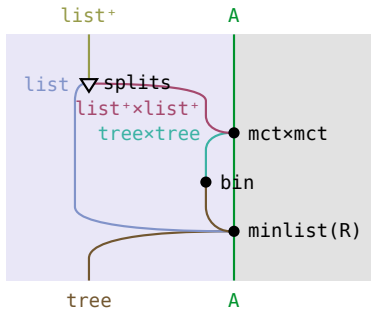
F(≤×𝟙)g⊑g≤ (9.4c)

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 — ≜ flatten	
⊑  definition of opb, and + monotonic	
⊑  ≤ reflexive, so zero⊑zero ≤; coproducts	
=  definition of g — ≜ flatten	

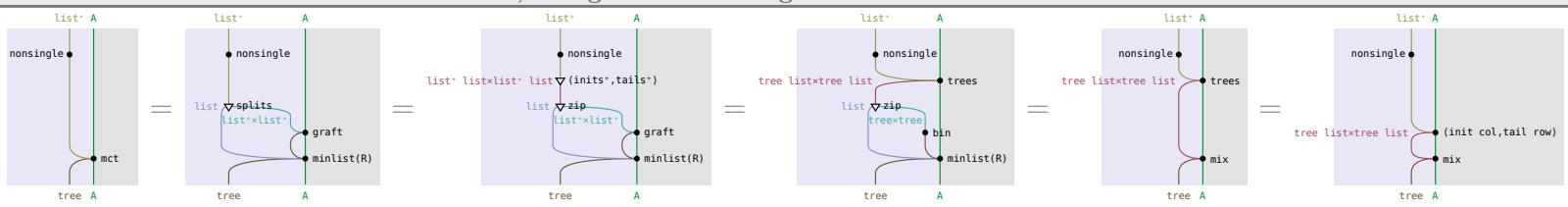
Assoc(sb) ⟹ (μX : $\frac{\text{[wrap,cat]°}}{\ni}$ P([tip,(X×X)bin])est(R))⊑$\frac{\text{flatten°}}{\ni}$ est(R) , mct=(single→head tip, ⟨init col,tail row⟩ mix) (9.4d)

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
list⁺(A) — $\frac{1}{\ni}$ — E(list⁺(A)) — E(flatten°) — E(tree(A)) — est(R) — tree(A) the specification — ≜ flatten	
list⁺(A) — $\frac{1}{\ni}$ — E(list⁺(A)) — E([wrap,cat]°) — EF(list⁺(A)) — P([tip,(X×X)bin]) — E(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)	
list⁺(A) — $\frac{1}{\ni}$ — E(list⁺(A)) — E(cat°) — E(list⁺(A)²) — P((X×X)bin) — E(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	
list⁺(A) — splits — [list⁺(A)²] — 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 (9.4e)

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**

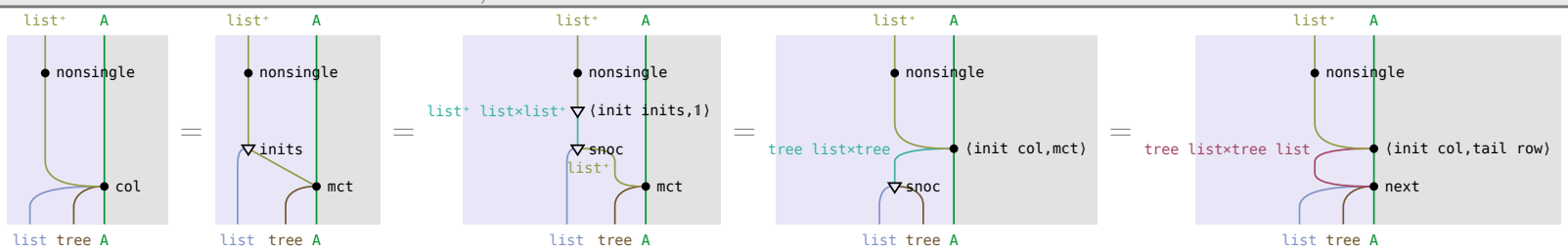


(a) splits — ≜ flatten (b) (c) (d)

(a) mct — ≜ flatten
(b) graft, trees; zip list(f×g)=(list(f)×list(g)) zip
(c) mix≜zip list(bin) minlist(R) — ≜ flatten
(d) inits⁺=init inits, tails⁺=tail tails; row, col — ≜ flatten

nonsingle col=nonsingle(init col,tail row)next (9.4f)

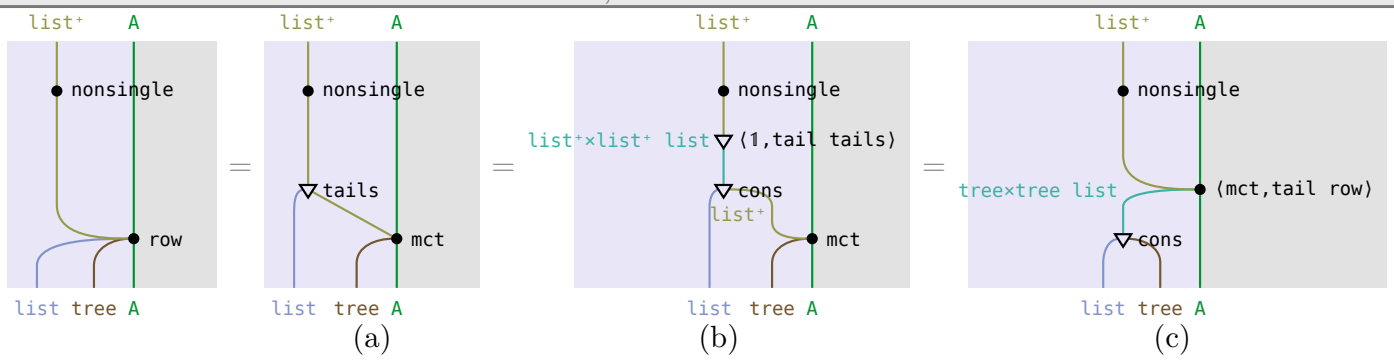
on a list of two or more elements, the column is what **next** makes of the column of its **init** and the row of its **tail**



(a) `col` — ≜ `flatten`
(b) `inits=⟨init inits,𝟙⟩ snoc` on non-singletons
(c) `snoc list(f)=(list(f)×f) snoc; col`
(d) (9.7) — (9.4e); `next≜⟨π₁,mix⟩ snoc` — ≜ `flatten`

nonsingle row=nonsingle(mct,tail row)cons (9.4g)

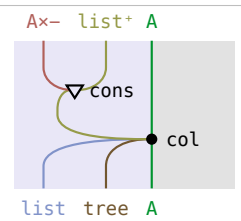
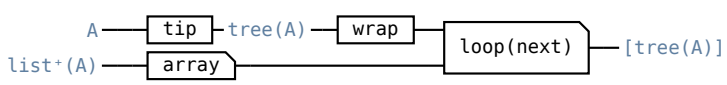
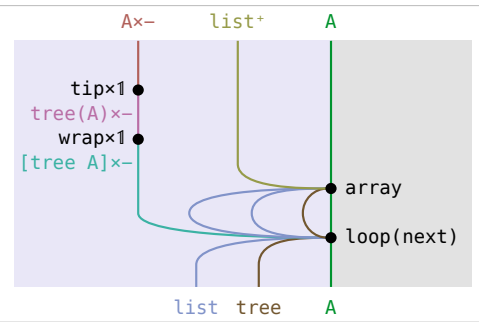
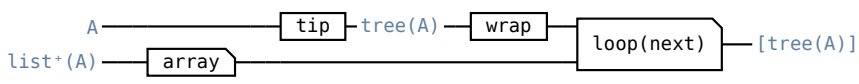
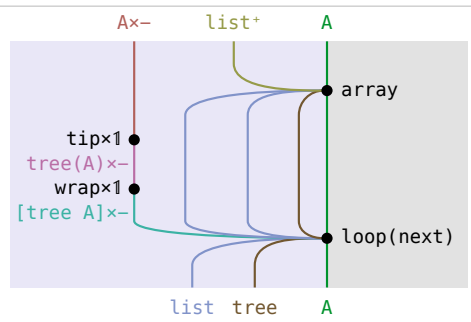
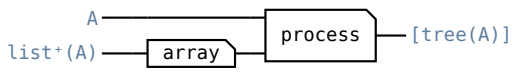
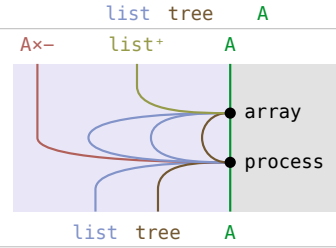
on a list of two or more elements, the row is **mct** of the whole list consed onto the row of its **tail**



(a) `row` — ≜ `flatten`
(b) `tails=⟨𝟙,tail tails⟩ cons` on non-singletons
(c) `cons list(f)=(f×list(f)) cons; row`

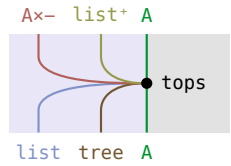
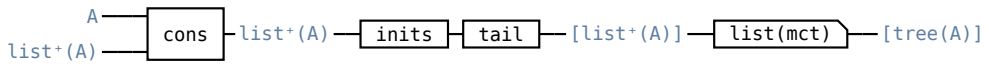
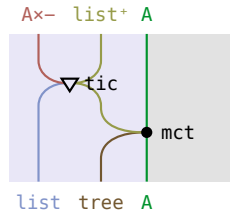
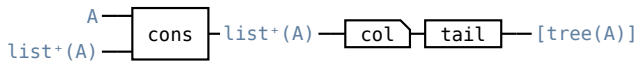
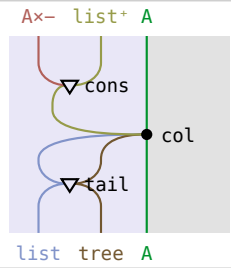
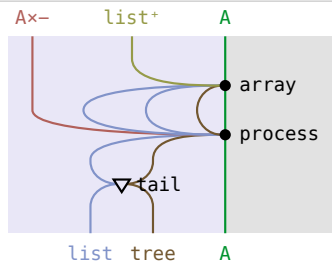
cons col=(𝟙×array)process (9.4h)

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) — (9.4f) — at `a:[b]` and at `a:(u++[b])`, with `[a] col=[tip(a)]`	
=  products	
=  definition of `process`	

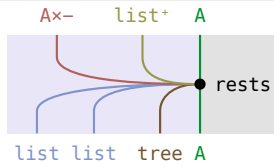
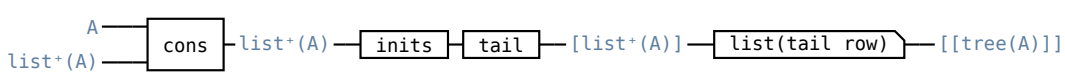
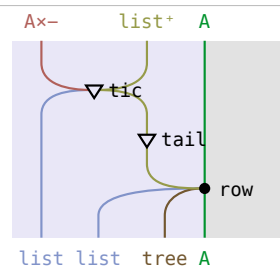
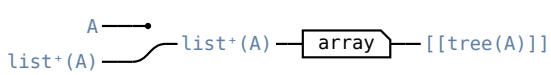
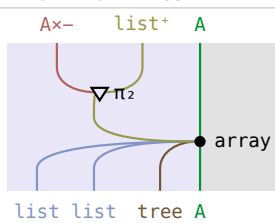
tops=(𝟙×array)process tail (9.4i)

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) — (9.4h)	

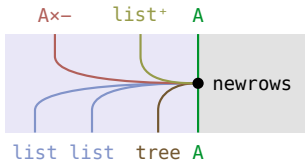
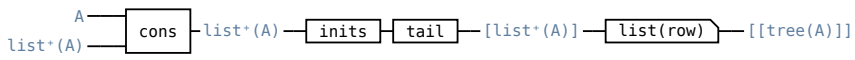
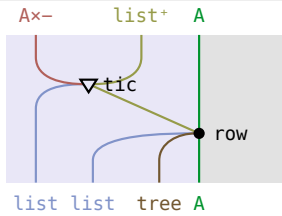
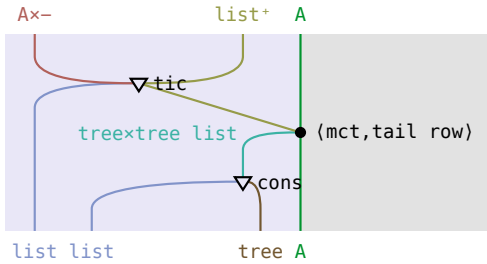
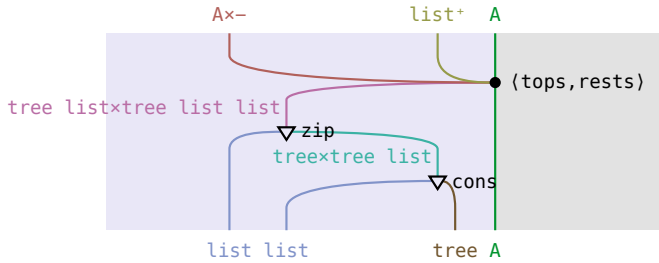
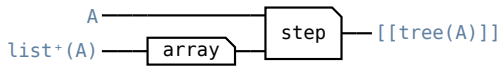
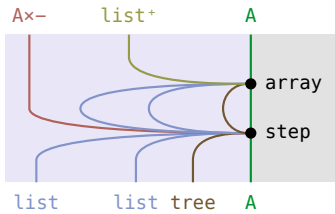
rests=π₂ array (9.4j)

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`	

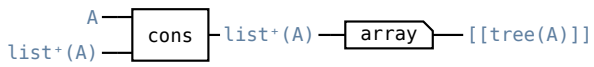
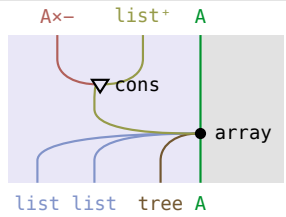
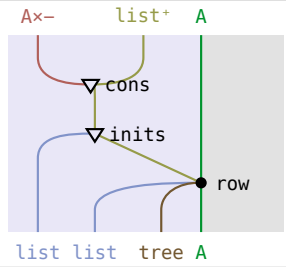
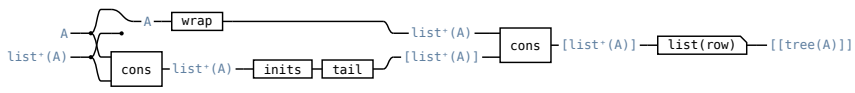
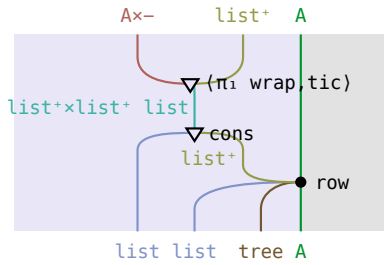
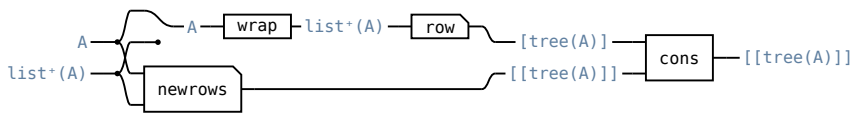
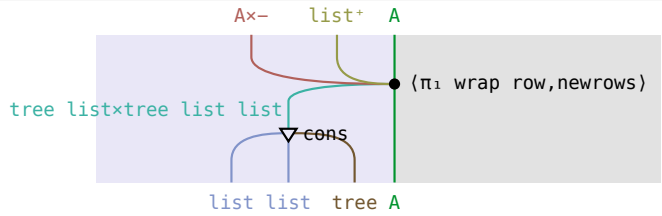
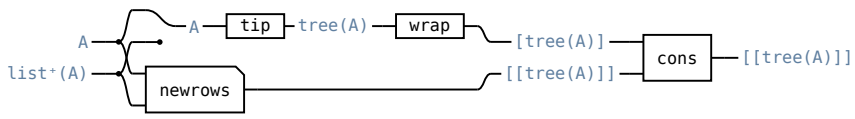
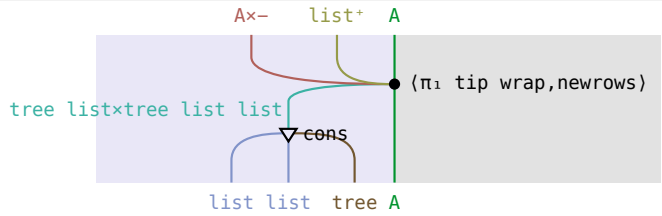
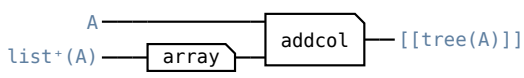
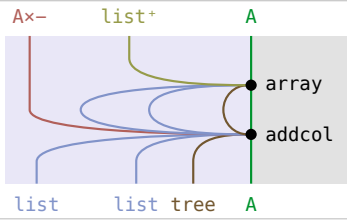
newrows=(𝟙×array)step (9.4k)

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) — (9.4g) — on the non-singleton `tic` lists	
= `list⟨f,g⟩=⟨list(f),list(g)⟩ zip`; products; definitions of `tops` and `rests`	
=  (9.4i) and (9.4j); products; definition of `step`	

cons array=(𝟙×array)addcol (9.4l)

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`	
=  (9.4k); products; definition of `addcol`	

§9.5. Data compression

(9.5a)

`[A]::=nil|snoc ([A],A)`, `list⁺ A::=wrap A|snoc (list⁺ A,A)`, `String=[Char]`, `Code::=sym Char|ptr (String,String⁺)`, `F(Code,String)=1+(String×Code)`, `α≜[nil,snoc]`.

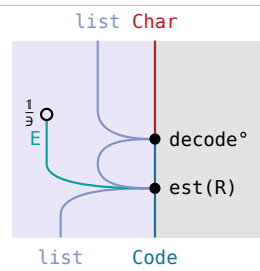
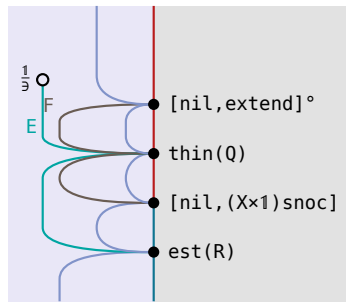
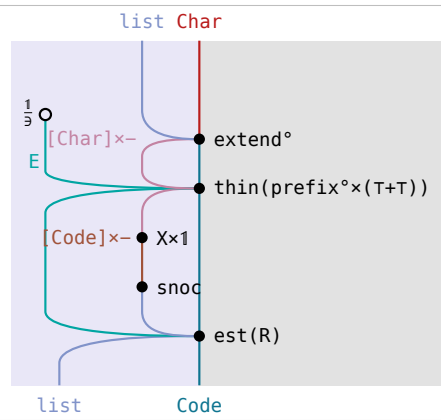
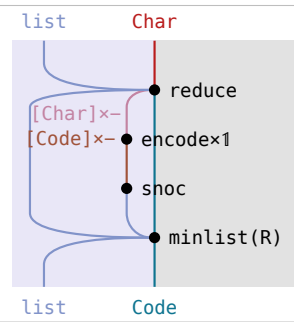
`decode≜⦇[nil,extend]⦈ : [Code]→String`, `extend (xs,sym a)=xs⧺[a]`, `extend (xs,ptr (ys,zs))=xs⧺zs` when `ys⧺zs` is a proper prefix of `xs⧺zs`; `H=decode°`.

`size≜⦇[zero,distr [𝟙×c,𝟙×p] plus]⦈` with `c`, `p` the constant costs of a symbol and a pointer; `R≜size≤size°`, `Q≜F(T+T,prefix°)=𝟙+(prefix°×(T+T))`, the two `T` on symbols and on pointers.

`lrt ws=est(prefix°×(T+T)) {(xs,(ys,zs))|xs⧺zs=ws, ys⧺zs` a proper prefix of `ws}`, the longest repeated tail; `reduce (ws⧺[a])=(zs≠[]→[(ws,sym a),(xs,ptr (ys,zs))],[(ws,sym a)])` with `(xs,(ys,zs))=lrt (ws⧺[a])`.

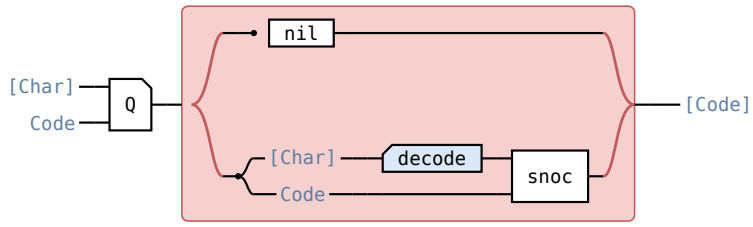
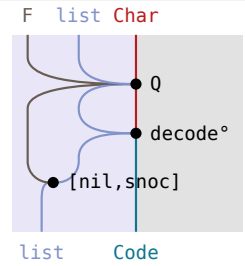
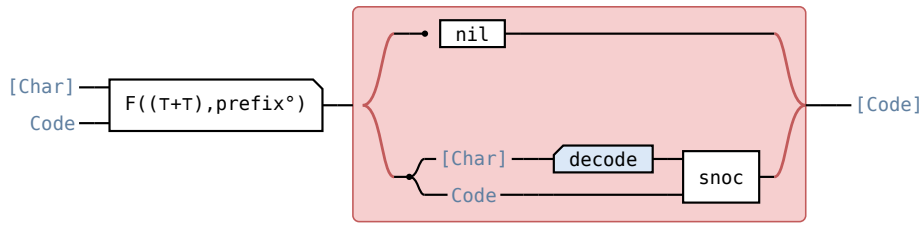
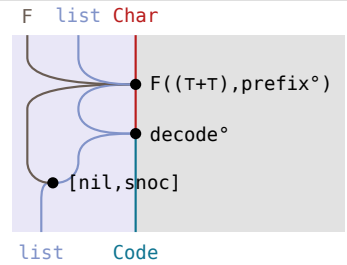
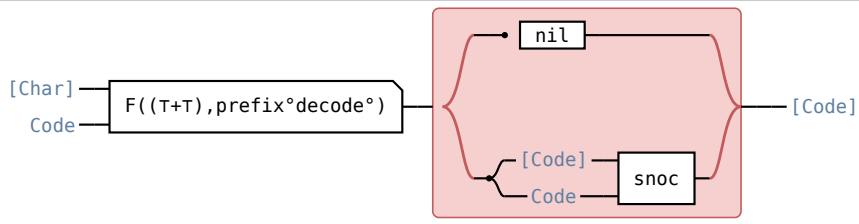
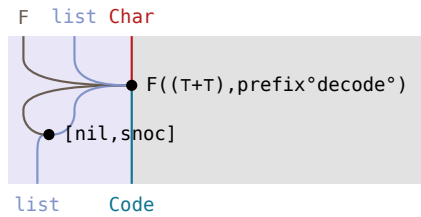
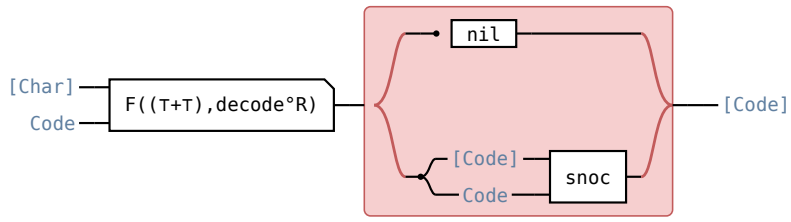
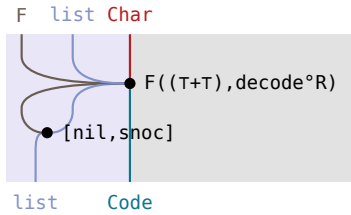
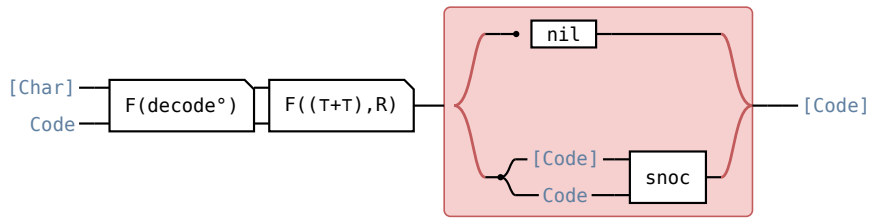
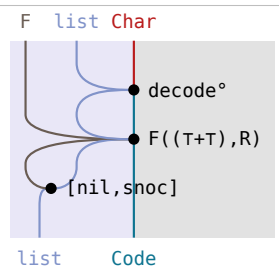
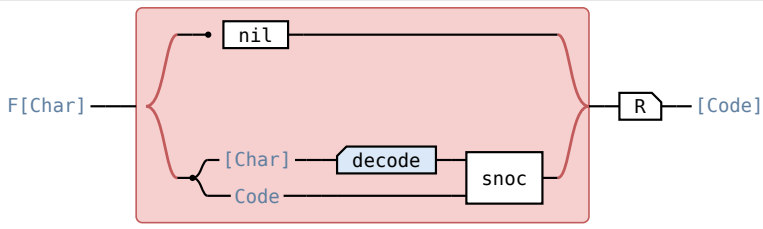
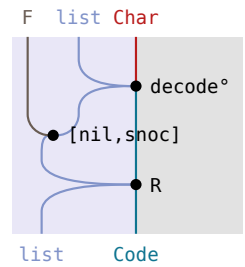
$$0 \le c \wedge 0 \le p \implies (\mu X : \frac{[nil,extend]^\circ}{\ni}\ thin(Q)P([nil,(X\times\mathbb{1})snoc])est(R)) \sqsubseteq \frac{decode^\circ}{\ni}\ est(R)\ , \qquad (9.5b)$$
$$encode=(null\to nil, reduce\ list((encode\times\mathbb{1})snoc)\ minlist(R))$$

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] — $\frac{1}{\ni}$ — E[Char] — E(decode°) — E[Code] — est(R) — [Code] the specification — $\triangleq$ decode	
[Char] — $\frac{1}{\ni}$ — E[Char] — E([nil,extend]°) — thin(Q) — EF[Char] — P([nil,(X×𝟙)snoc]) — E[Code] — est(R) — [Code] at Q≜F(T+T,prefix°)=𝟙+(prefix°×(T+T)), the two T on symbols and on pointers. Monotonicity F(R)α⊑αR 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)α⊑αR, and decode prefix⊑R° decode, for which decode init⊑R° 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] — $\frac{1}{\ni}$ — E[Char] — E(extend°) — thin(prefix°×(T+T)) — E([Char]×Code) — P((X×𝟙)snoc) — E[Code] — est(R) — [Code] = Proposition 9.1: nil and extend have disjoint ranges. The decompositions of one string are $\frac{extend^\circ}{\ni}$ (ws#[a])={(ws,sym a)} ∪ {(xs,ptr (ys,zs))|xs#zs=ws#[a], ys#zs a proper prefix of 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 $\frac{extend^\circ}{\ni}$ thin(prefix°×(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	

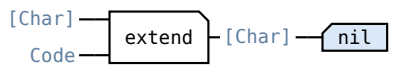
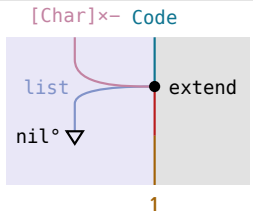
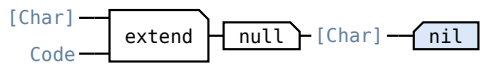
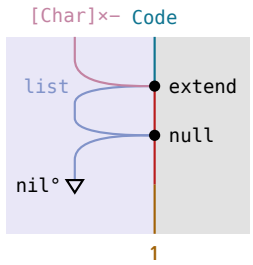
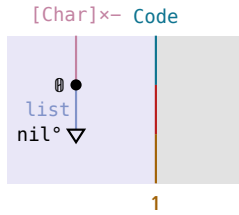
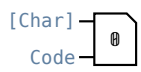
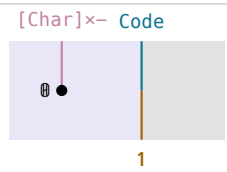
0≤c ∧ 0≤p ⟹ QF(decode°)[nil,snoc]⊑F(decode°)[nil,snoc]R (9.5c)

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°) — ≜ decode	
=  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` (9.5d)

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

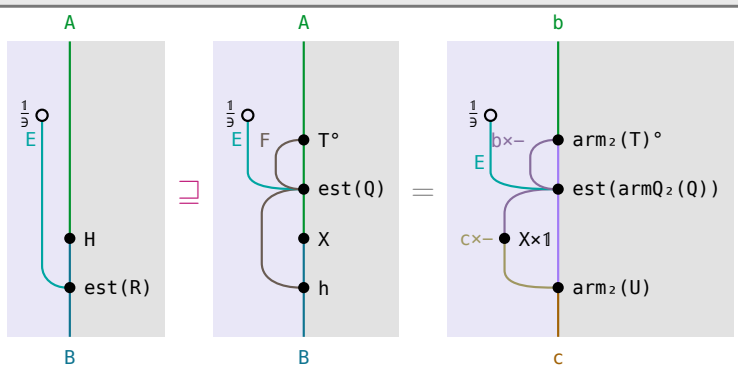
§10.1. Theory

(10.1a)

h, T, R, H, M as in ≜ H; 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) ∧ MonoAlg(h,R) ∧ RR⊑R ∧ ThinCondition(T,h,R,Q) ⟹ (μX : $\frac{T^\circ}{\ni}$ est(Q)F(X)h)⊑$\frac{H}{\ni}$ est(R) (10.1b)

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

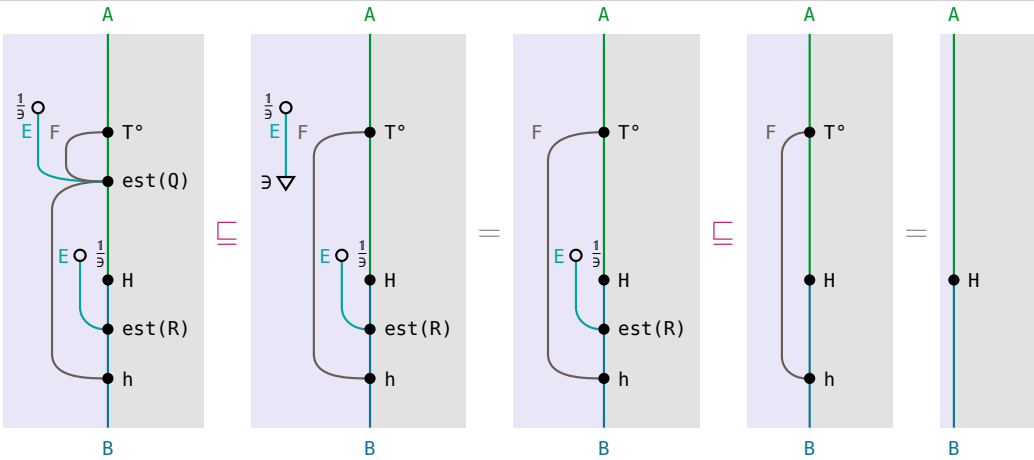


(a) (b)

(a) H≜⦇T⦈°⦇h⦈ — ≜ H, Q
(b) Proposition 10.1, V₂V₁°=𝟘

T°F(H)h=H ⟹ $\frac{T^\circ}{\ni}$ est(Q)F($\frac{H}{\ni}$ est(R))h⊑H (10.1c)

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

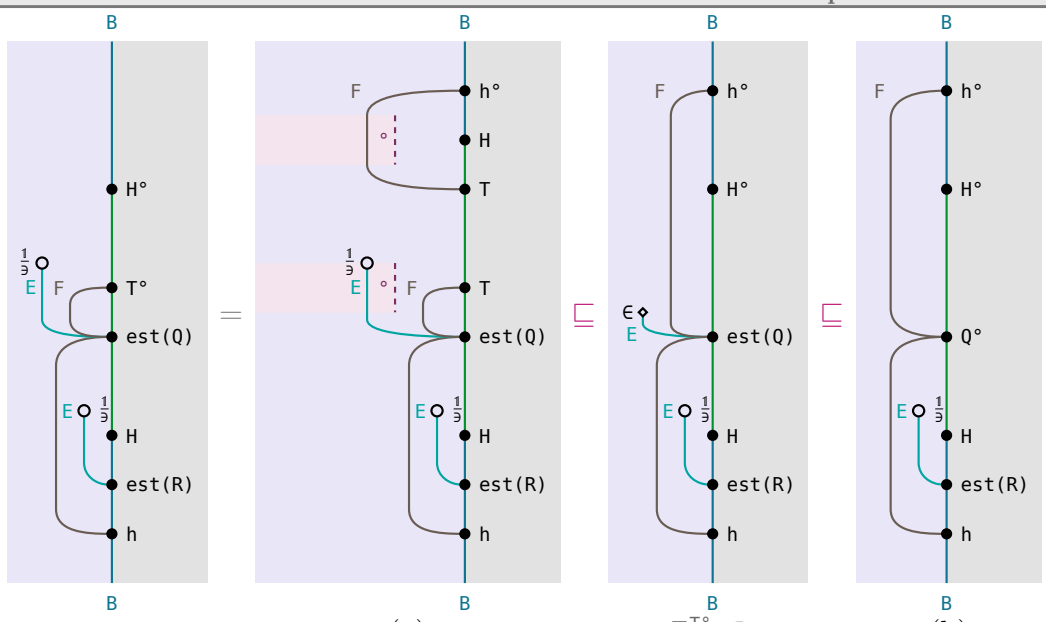


(a) (b) (c) (d)

(a) est(Q)⊑∋ — ≜ est
(b) $\frac{T^\circ}{\ni}$∋=T° — (3.1j)
(c) M⊑$\frac{H}{\ni}$∋=H — (7.1.1a)
(d) T°F(H)h=H — hylomorphism theorem

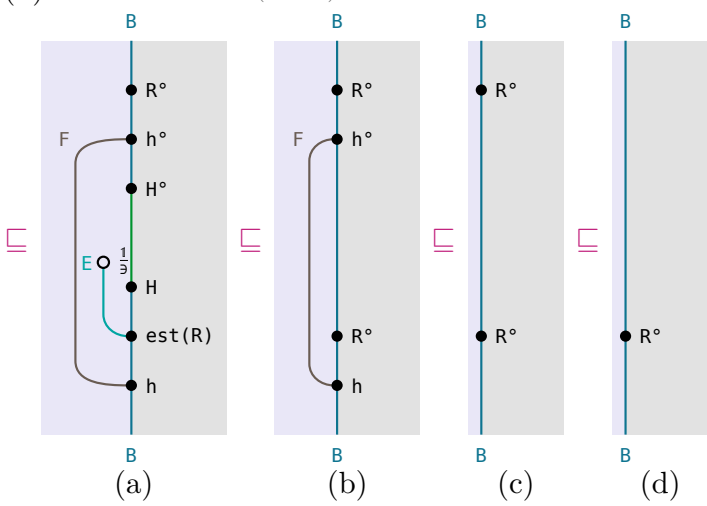
$$\mathtt{Map(h)} \wedge \mathtt{MonoAlg(h,R^\circ)} \wedge \mathtt{R^\circ R^\circ \sqsubseteq R^\circ} \wedge \mathtt{T^\circ F(H)h=H} \wedge \mathtt{QF(H)h \sqsubseteq F(H)hR} \Longrightarrow \mathtt{H^\circ} \tfrac{\mathtt{T^\circ}}{\ni}\ \mathtt{est(Q)F(}\tfrac{\mathtt{H}}{\ni}\ \mathtt{est(R))h \sqsubseteq R^\circ} \qquad (10.1d)$$

H° followed by the greedy body at M is $\sqsubseteq R^\circ$: an answer of the body is never worse than an answer of H to the same input



$T\frac{T^\circ}{\ni} \sqsubseteq \in$

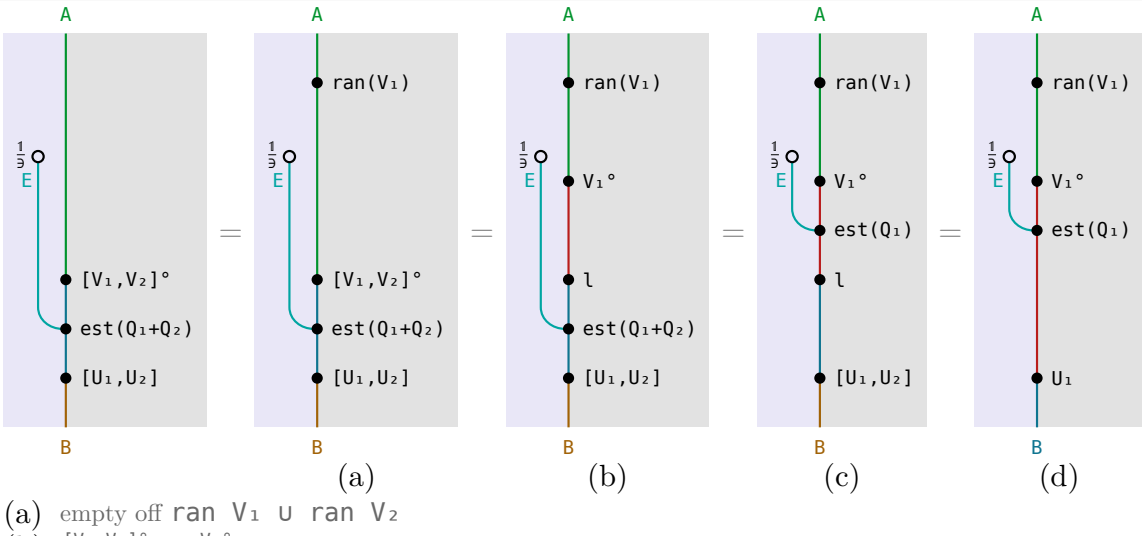
(a) $H^\circ = h^\circ F(H^\circ)T$ — hylomorphism theorem
(b) $\in est(Q) \sqsubseteq Q^\circ$ — (7.1.1a)



(a) $QF(H)h \sqsubseteq F(H)hR$ conversed
(b) $H^\circ M \sqsubseteq R^\circ$ under F — (7.1.1a)
(c) $h^\circ F(R^\circ)h \sqsubseteq R^\circ$
(d) R transitive

$$V_2\ V_1^\circ = \emptyset \Longrightarrow \tfrac{[V_1,V_2]^\circ}{\ni}\ est(Q_1+Q_2)[U_1,U_2] = ran(V_1)\tfrac{V_1^\circ}{\ni}\ est(Q_1)U_1 \cup ran(V_2)\tfrac{V_2^\circ}{\ni}\ est(Q_2)U_2 \qquad (10.1e)$$

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



(a) empty off $ran\ V_1 \cup ran\ V_2$
(b) $\frac{[V_1,V_2]^\circ}{\ni} = \frac{V_1^\circ}{\ni} P(inl)$ on $ran\ V_1$ — (9.2i)
(c) $P(inl)est(Q_1+Q_2) = est(Q_1)inl$
(d) $inl[U_1,U_2] = U_1$

§10.2. The detab-entab problem

(10.2a)

`detab≜⦇[nil,expand]⦈ : String→String` over snoc-lists, `α≜[nil,snoc]`, `H=detab°`; `expand (xs,a)=(a=TB→fill xs,xs#[a])`, `fill xs=xs#blanks (n−(col xs) mod n)`.

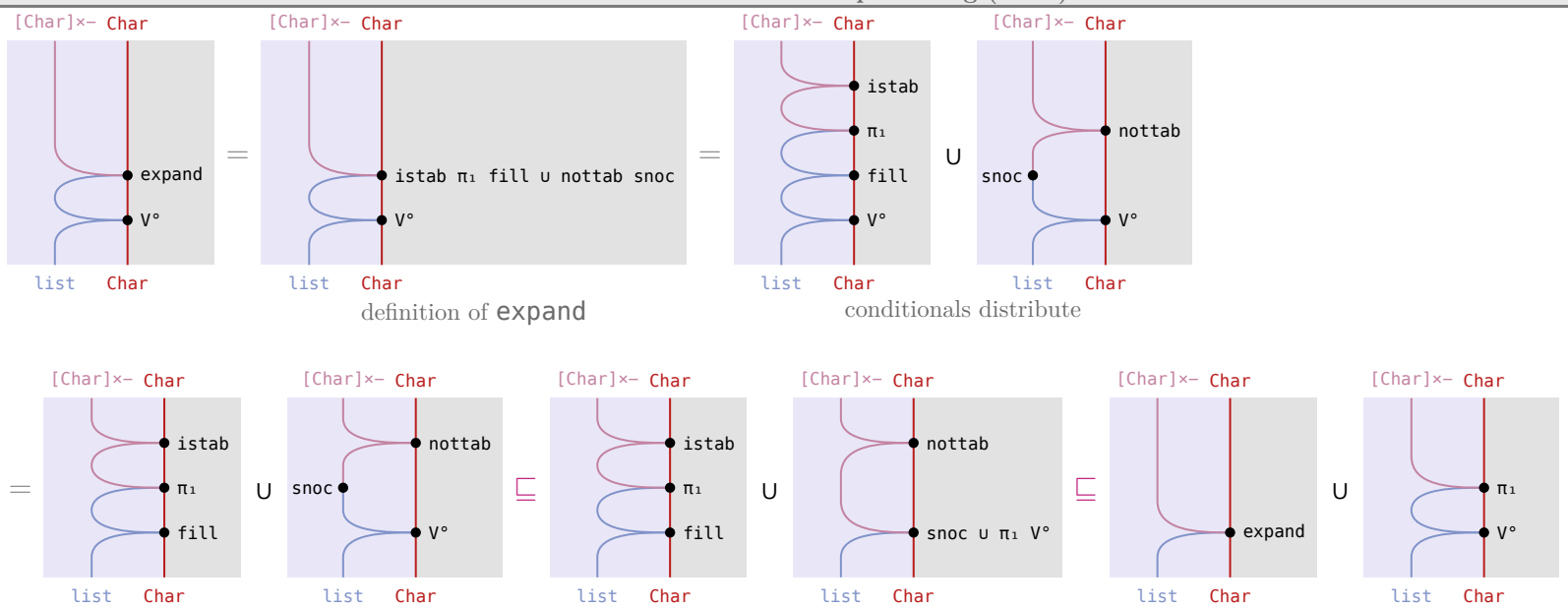
`col≜⦇[zero,count]⦈`, `count (c,a)=(a=NL→0,c+1)`; `TB` the tab, `BL` the blank, `NL` the newline, tab stops every `n` columns.

`R≜length≤length°` , `U` the preorder with `a U b⟺a=TBva=b`, `V≜prefix°n(fill fill°)` , `Q≜𝟙+(V×U)` .

`unfill xs` the shortest prefix of `xs` with `fill (unfill xs)=fill xs`; `tbc` the trailing blank count, `triple≜⟨unfill entab,⟨tbc,col⟩⟩`.

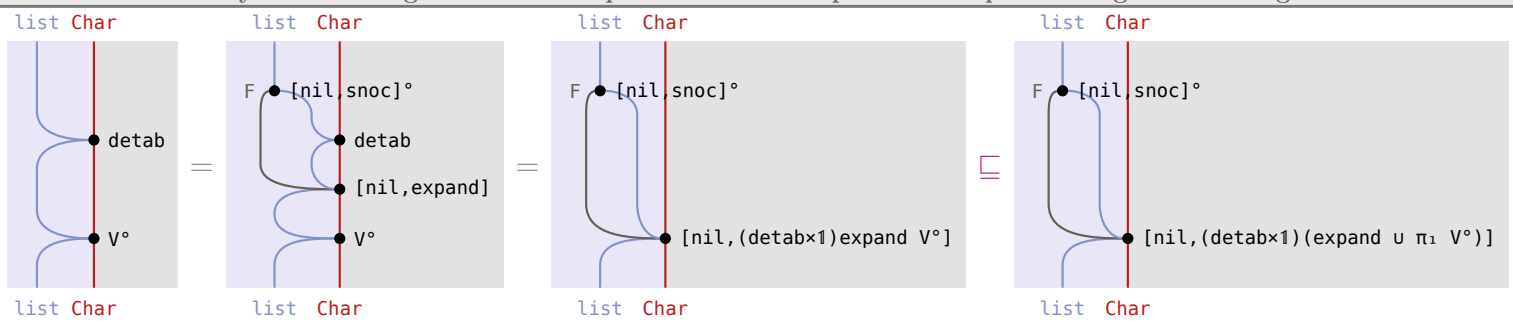
`0 < n ∧ blank ≠ nl ⟹ expand V°⊑expand ∪ π₁ V°` (10.2b)

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°**)



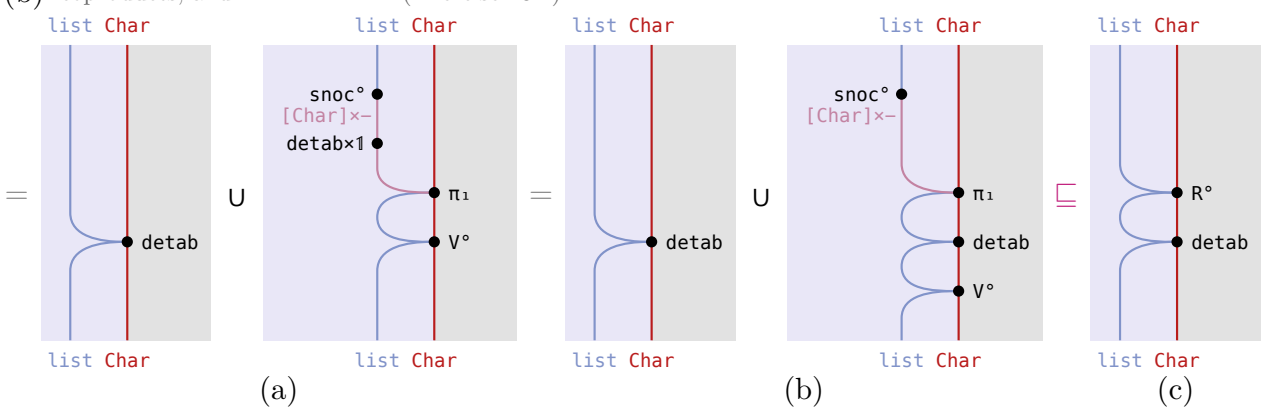
`0 < n ∧ blank ≠ nl ⟹ detab V°⊑R°detab` (10.2c)

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



(a) `detab` is a fold: `detab=α°F(detab)[nil,expand]` — ≜ `detab`

(b) coproducts, and `nil V°=nil` (Exercise 10.4)



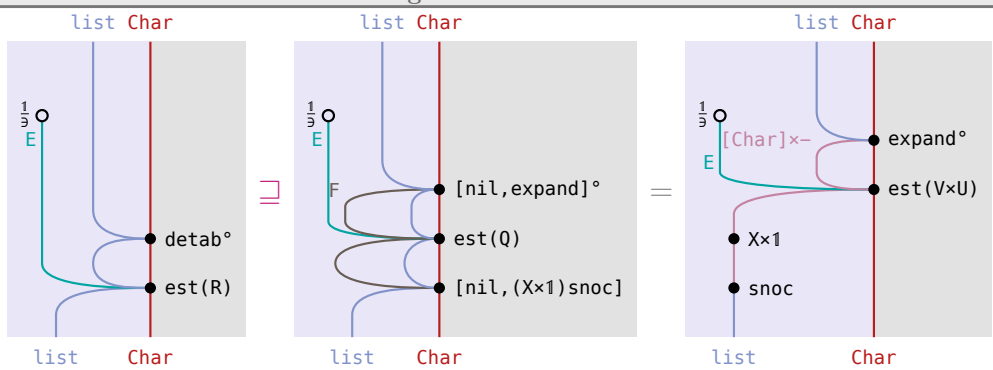
(a) distributing `∪`; the fold again, and the definition of `F`

(b) naturality of `π₁`: `(detab×𝟙)π₁=π₁ detab`; `snoc°π₁` is `init`

(c) `init` is inductive, so `X≜detab V°`, a solution of `X⊑detab∪(init X)`, lies below the greatest one, `prefix detab` (induction on the input); `prefix⊑R°`

$$0 < n \wedge blank \neq nl \Longrightarrow (\mu X : \frac{[nil,expand]^\circ}{\ni} est(Q)F(X)[nil,snoc]) \sqsubseteq \frac{detab^\circ}{\ni} est(R) \qquad (10.2d)$$

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



(a) (b) (c)

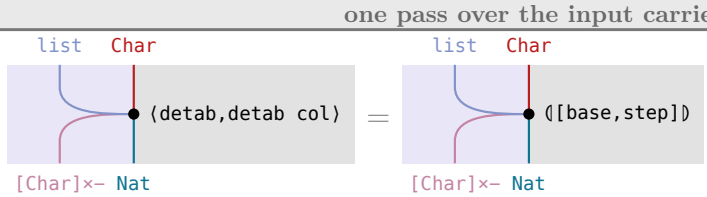
(a) the specification — $\triangleq$ `detab`; `detab entab=𝟙` and nothing shorter does

(b) 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

(c) 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

$$blank \neq nl \wedge tb \neq nl \Longrightarrow (detab,detab\ col)=(\!|[base,step]|\!) \qquad (10.2e)$$

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



(a)

(a) 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

(10.2f)

- $blank \neq nl \wedge tb \neq nl \Longrightarrow (\!|[base,step]|\!)\ (convert\ l)\ r \Longleftrightarrow r=loop(step,(([],0),l))$

  the fold of `[base,step]` over the input converted to a snoc-list is the left loop of `step` from `base`

- $\pi_1(loop(step,(([],\ c_0),\ x)))=loop'((f,\ g),(c_0,x))$

  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

(10.2g)

- $a \neq tb \Longrightarrow expand(y,b)=snoc(x,a) \Longleftrightarrow fill(y)=snoc(x,a) \wedge b=tb \vee (y,b)=(x,a)$

  `expand` produces `x♯[a]` from a tab after a string that fills to it, or from `x` and `a`

- $0 < n \wedge blank \neq nl \Longrightarrow (\exists y.\ fill(y)=snoc(x,a)) \Longleftrightarrow a=blank \wedge col\ (snoc\ x\ a)\ \%\ n=0$

  a string is a `fill` exactly when it ends in a blank on a tab stop

(10.2h)

- $entab(x)=entab\ (unfill\ x) \mathbin{++} blanks\ (tbc\ x)$

  (10.1): the output of `entab` is that of `unfill x` followed by the trailing blanks held back

- $tbc[]=0$

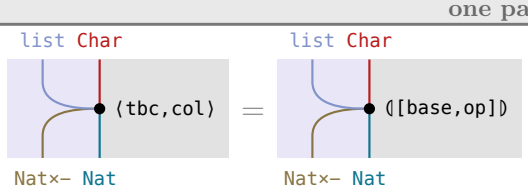
  the empty string has no trailing blanks

- $tbc(snoc(x,a))=if\ a = blank \wedge col\ (snoc\ x\ a)\ \%\ n \neq 0\ then\ tbc\ x + 1\ else\ 0$

  a blank off a tab stop adds one held blank; anything else releases them all

$$blank \neq nl \Longrightarrow (tbc,col)=(\!|[base,op]|\!) \qquad (10.2i)$$

one pass counts the held blanks and the column together

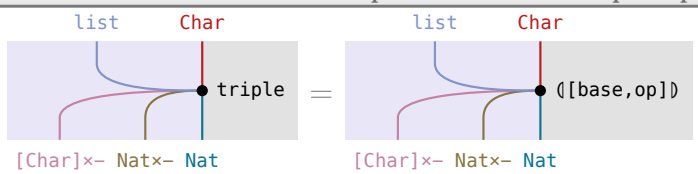


(a)

(a) `op` on `(tbc x,col x)` is `(tbc,col)` one character further, from the equations for `tbc` — (10.2h)

blank ≠ nl ⟹ triple=⦇[base,op]⦈ (10.2j)

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

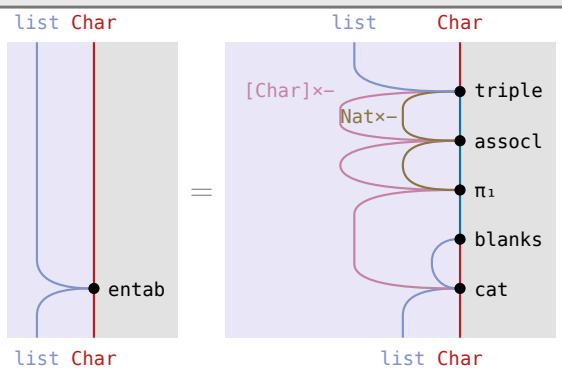


(a)

(a) `op` on `triple x` is `triple` one character further: (10.1) where the blanks are released — (10.2h)

entab=triple assocl π₁ (𝟙×blanks)cat (10.2k)

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



(10.1) — (10.2h)

§10.3. The minimum tardiness problem

(10.3a)

`FX=1+(X×Job)`, `α≜[nil,snoc]` on schedules, `β≜[nil,snag]` on bags, `snag` putting a job into a bag; `bagify≜⦇β⦈ : [Job]→Bag Job`, `H=bagify°`.

`ct`, `dt`, `wt : Job→Real` the completion, due and weighting quantities of a job; `penalty(xs,j)=(sum(list(ct)(xs))+ct(j)−dt(j))×wt(j)`.

`cost≜` $\frac{\texttt{prefix}}{\ni}$ `P(α° [zero,penalty]) est(≥)`, `cost []=0`, `cost (xs⧺[j])=bmax (cost xs,penalty (xs,j))`, `R≜cost≤cost°`.

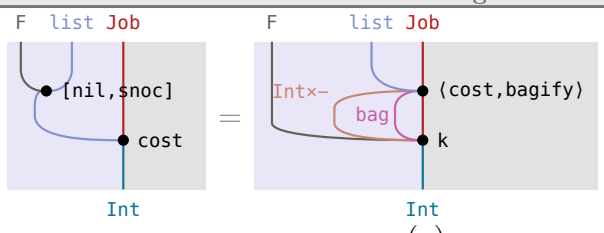
`perm≜bagify bagify°=⦇[nil,add]⦈`, `add (xs,j)=ys⧺[j]⧺zs` for some `xs=ys⧺zs`.

`k≜[zero,assocr (𝟙×((bagify°×𝟙) penalty)) bmax]`, `f≜[zero,(bagify°×𝟙) penalty]`, `Q≜f≤f°`, `Q'≜(bagify°×𝟙) penalty≤penalty°(bagify×𝟙)`.

`g≜[zero,penalty]`, `m≜[zero,π₁ cost]` (B&dM's `h`, renamed as in (10.3k)).

[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

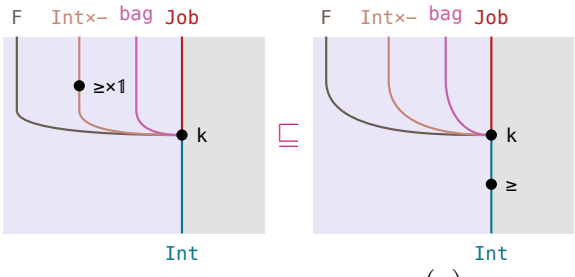


(a)

(a) `cost (xs⧺[j])=bmax (cost xs,penalty (xs,j))`, and `penalty(xs,j)` reads only the bag of `xs` — ≜ `bagify`

F(≥×𝟙)k⊑k≥ (10.3c)

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

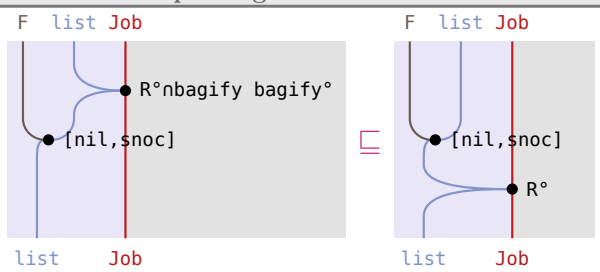


(a)

(a) `bmax` is monotone in its first argument

F(R°nbagify bagify°)[nil,snoc]⊑[nil,snoc]R° (10.3d)

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

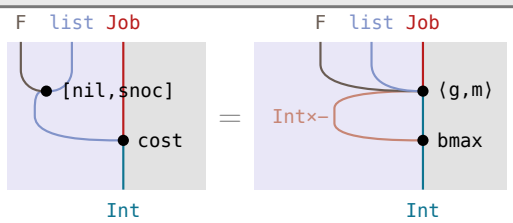


(a)

(a) Proposition 9.3 at S≜bagify, ≤≜≥ — (9.2l), with (10.3b) and (10.3c)

[nil,snoc]cost=⟨g,m⟩bmax (10.3e)

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

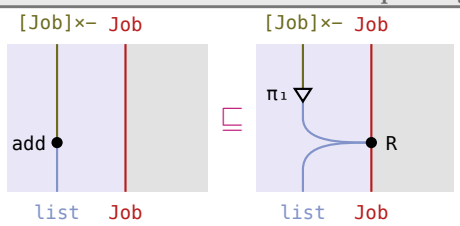


(a)

(a) cost (xs⧺[j])=bmax (cost xs,penalty (xs,j)) and bmax commutes — ≜ bagify

(∀j. 0≤ct(j)) ∧ (∀j. 0≤wt(j)) ⟹ add⊑π₁ R (10.3f)

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

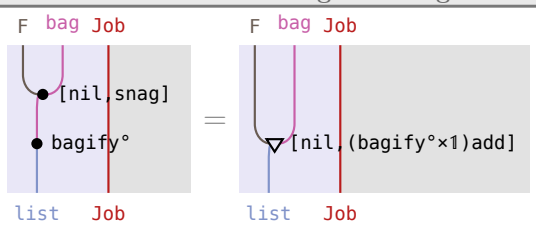


(a)

(a) induction on where j goes: every job after it starts later, so no penalty falls

[nil,snag]bagify°=[nil,(bagify°×𝟙)add] (10.3g)

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

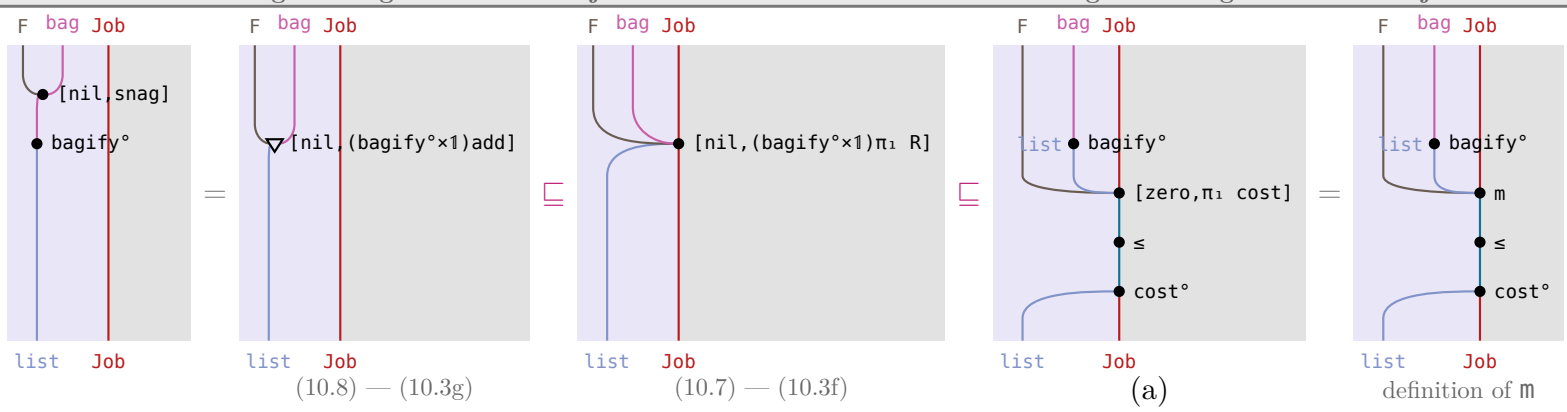


(a)

(a) take out the last copy of j and put it back — ≜ bagify

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

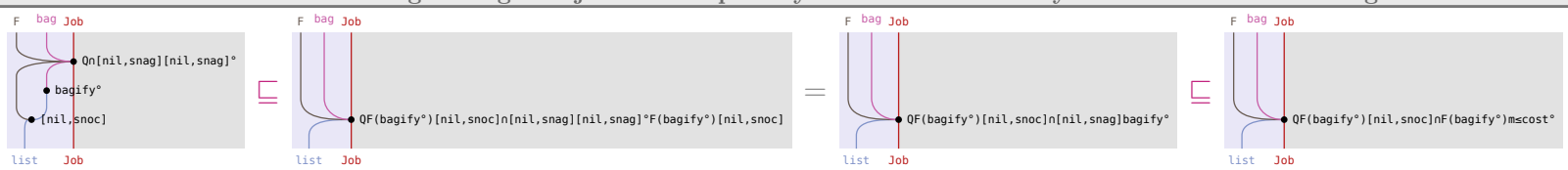
an ordering of a bag with one more job costs at least the cost of the ordering of the bag before its last job



(a) definition of R, and nil⊑zero≤cost°

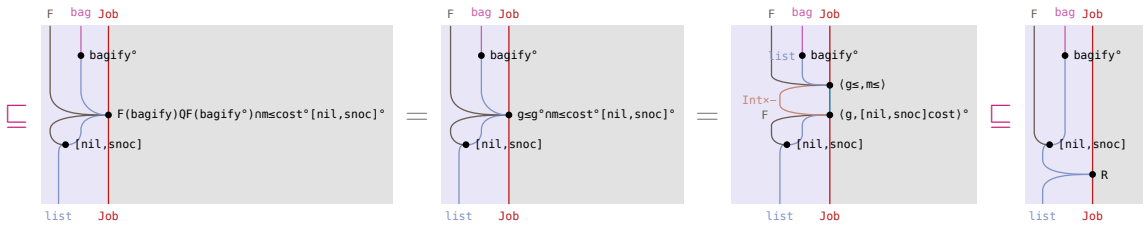
(∀j. 0≤ct(j)) ∧ (∀j. 0≤wt(j)) ⟹ (Q∩[nil,snag][nil,snag]°)F(bagify°)[nil,snoc]⊑F(bagify°)[nil,snoc]R (10.3i)

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



monotonicity of composition (a) (10.3h)

(a) β°F(bagify°)α=bagify°, since bagify=⦇β⦈

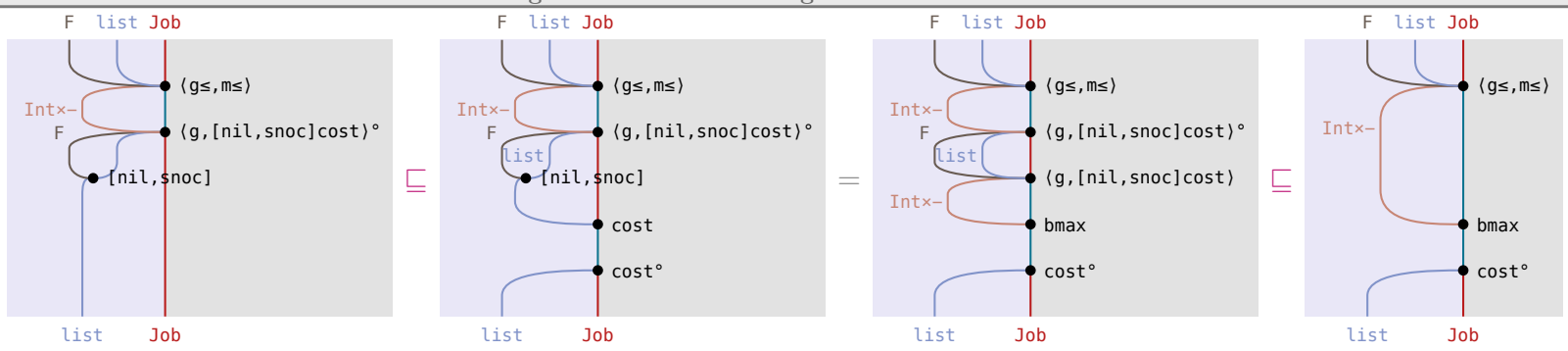


modular law (a) (b) (10.3j)

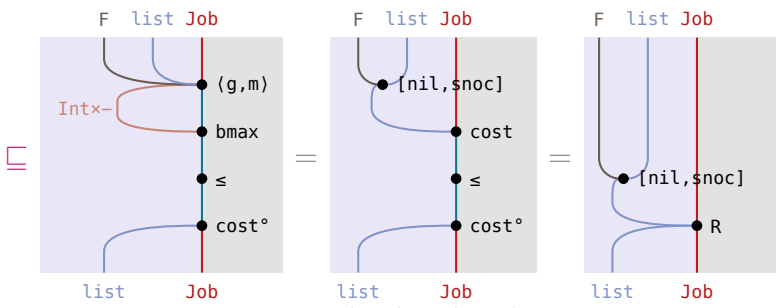
(a) choice of Q: F(bagify) Q F(bagify°)=g≤g°
(b) products: ⟨R,S⟩⟨T,U⟩°=RT°∩SU°

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

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



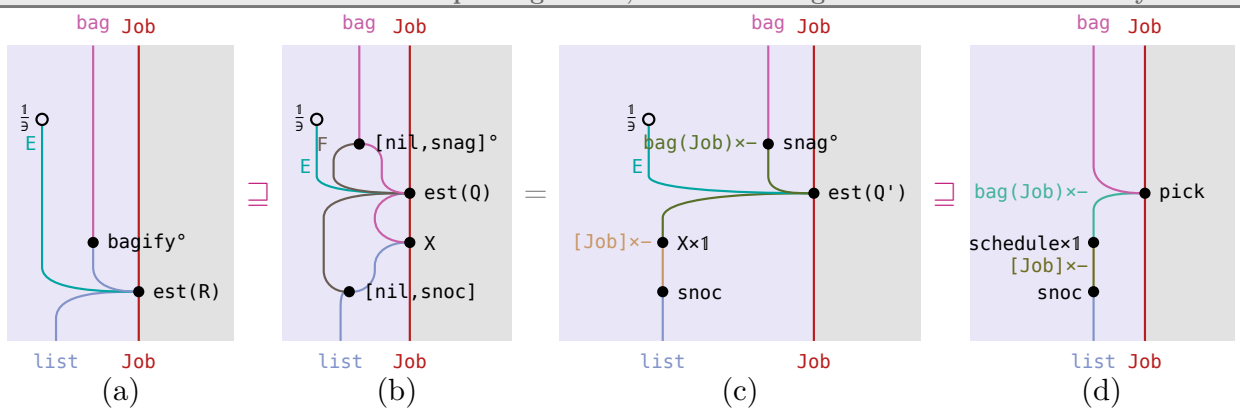
cost a map, so 𝟙⊑cost cost° α cost=⟨g,α cost⟩ bmax ⟨g,α cost⟩ simple



monotonicity of bmax (10.4) — (10.3e) definition of R

(∀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) , schedule=(null→nil,pick (schedule×𝟙) snoc) (10.3k)

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



(a) (b) (c) (d)

(a) the specification — ≜ bagify
(b) 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
(c) Proposition 10.1: nil and snag have disjoint ranges
(d) pick⊑$\frac{\text{snag°}}{\ni}$ est(Q'), a partial function, quadratic in the number of jobs

§10.4. The TeX problem

(10.4a)

`intern≜val round : Decimal→[0,2¹⁶)`, `val≜⦇[zero,shift]⦈`, `shift (d,r)=(d+r)/10`, `round r` rounds `2¹⁶r` to the nearest integer: `round r=n⟺2n−1<2¹⁷r<2n+1`.

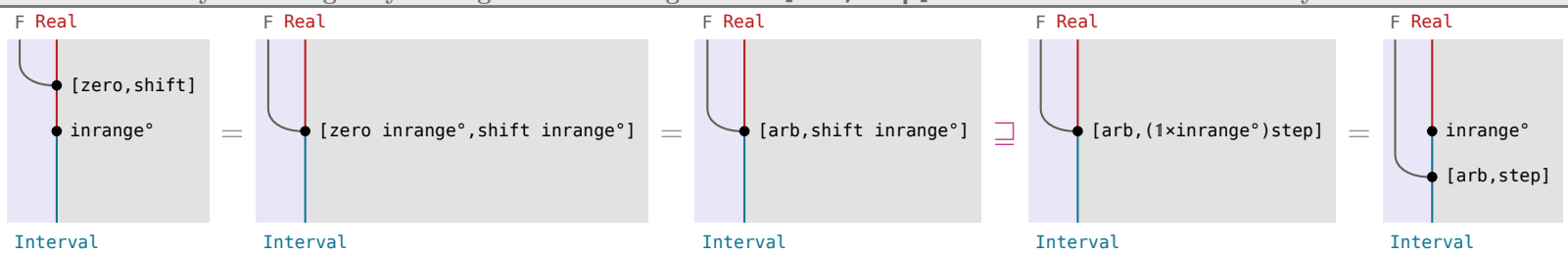
`interval n=((2n−1)/2¹⁷,(2n+1)/2¹⁷)`, `r inrange (a,b)⟺a<r<b`, `round°=interval inrange`, `R≜length≤length°`.

`Interval` the pairs `(a,b)` with `Legal(a,b)⟺0<b<1` and `a<b` (10.9); `step_legal` is what types `step [arb,step] : 1+(Digit×Interval)→Interval`, `step (d,(a,b))=((d+a)/10,(d+b)/10)`.

`FX=1+(Digit×X)`, `α≜[nil,cons]`, `H≜⦇[arb,step]⦈°`, `! : Digit×Interval→1`, `Q≜(l°!°r) ∪ 𝟙` `l`, `r` are (5.3a)'s injections into `FX=1+(Digit×X)`, so `l : 1→FX` and `r : Digit×X→FX`, `w≜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

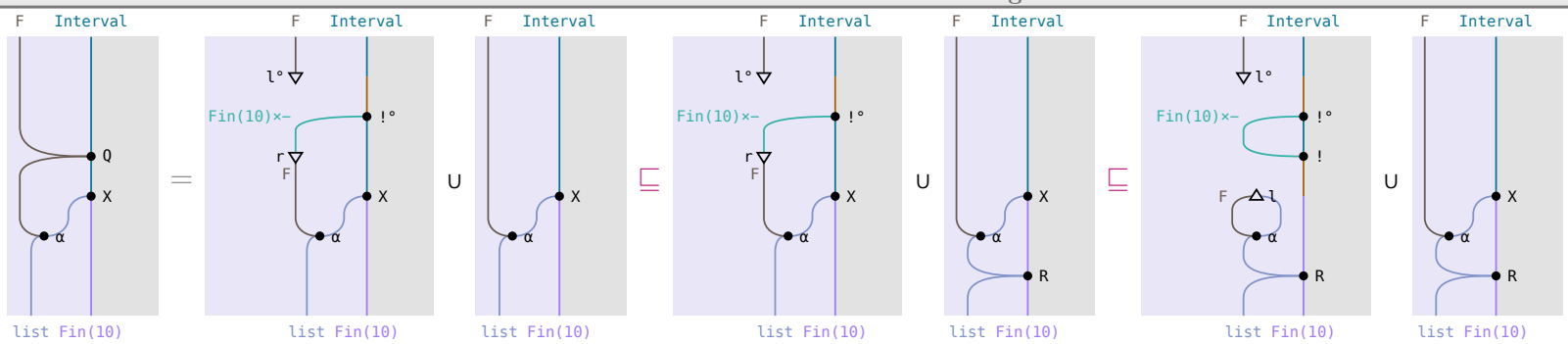


(a) (b) (c) (d) (e)

(a) fusion: it suffices that `[zero,shift] inrange°⊒F(inrange°)[arb,step]`
(b) `[T,U]Z=[TZ,UZ]`, a coproduct law not tabulated in the note
(c) `zero inrange°=arb`: the first condition, which determines `arb`
(d) arithmetic: `10a−d<r<10b−d ⟹ a<(d+r)/10<b` for `(a,b)=step(d,(10a−d,10b−d))`; only ⊒, since `(10a−d,10b−d)` satisfies (10.9) only when `d<10b<d+1`
(e) `F(S)[T,U]=[T,(𝟙×S)U]` read right to left: definition of `F`

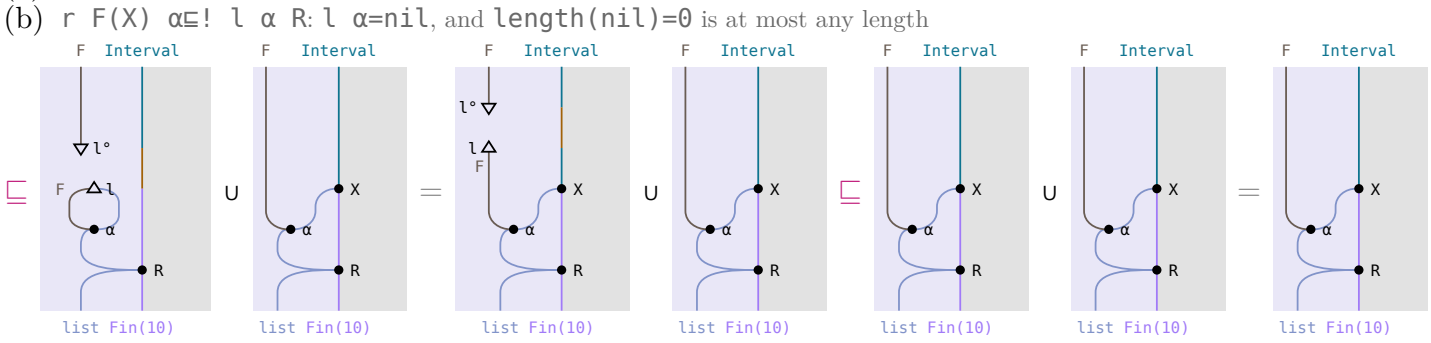
`QF(X)α⊑F(X)αR` (10.4c)

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



(a) `R` is reflexive (b)

(a) definition of `Q`; composition distributes over `∪`
(b) `r F(X) α⊑! l α R`: `l α=nil`, and `length(nil)=0` is at most any length

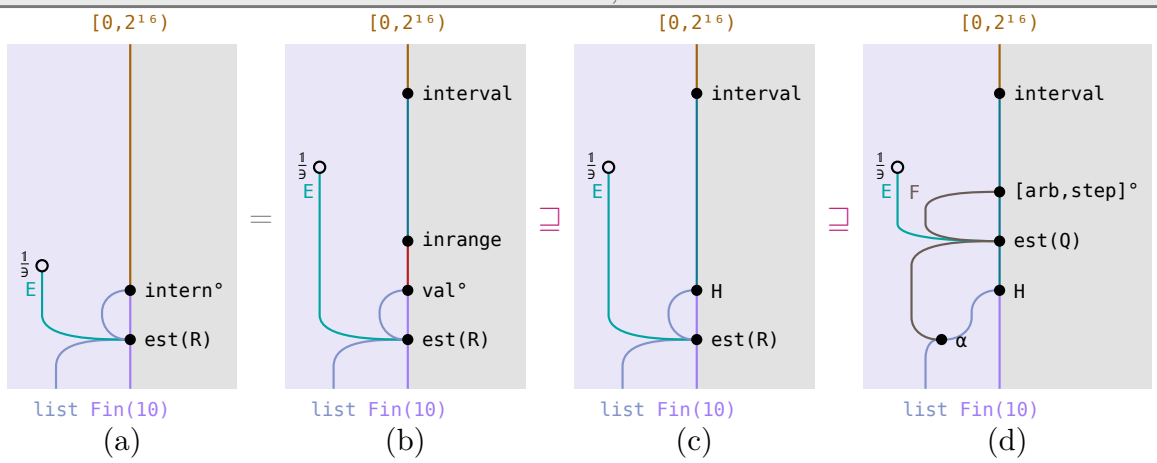


(a) `l F(X)=l`: definition of `F` `l` is simple: `l°l⊑𝟙` (b)

(a) `!°!⊑𝟙` on `𝟙`: universal property of `!`
(b) `∪` is idempotent

$$\texttt{interval}(\mu X : \tfrac{\texttt{[arb,step]}^\circ}{\ni}\ \texttt{est(Q)F(X)}\alpha) \sqsubseteq \tfrac{\texttt{intern}^\circ}{\ni}\ \texttt{est(R)}\ ,\quad \texttt{extern(n)=f(2n-1,2n+1)} \tag{10.4d}$$

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



(a) the specification — $\triangleq$ `intern`
(b) `round°` is not a map, but `interval` is, so it comes out of the transpose
(c) the type restriction (10.9): a shortest decimal `H` gives an interval is a shortest one among all decimals inside it, and is inside it by (10.4b)
(d) $\frac{\texttt{[arb,step]}^\circ}{\ni}$ returns at most two elements — stop, or take one more digit — and `! nil⊑cons R°` makes it stop whenever stopping is legal

(10.4e)

- `f(a,b)=x ⟺ a < 0 ∧ x=[] ∨ ¬a < 0 ∧ (∃d. ∃y. d = ⌊10b⌋ ∧ f(10a−d,10b−d)=y ∧ x=[d]+y)`
  `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)`
- $(\mu X : \frac{\texttt{[arb,step]}^\circ}{\ni}\ \texttt{est(Q)F(X)}\alpha)\texttt{=f}$
  the least solution of the greedy recursion of (10.4d) 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 ∧ d=10q/w ⟹ f(p,q)=[d]+f(10p−wd,10q−wd)`
  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)`

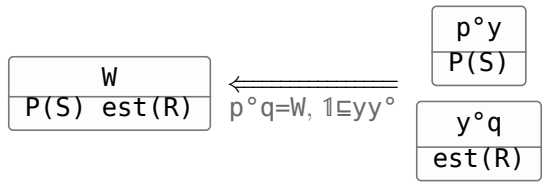
§11. Appendix

§11.1. P(S) est(R)=(∋S)∩(∈\(SR°))

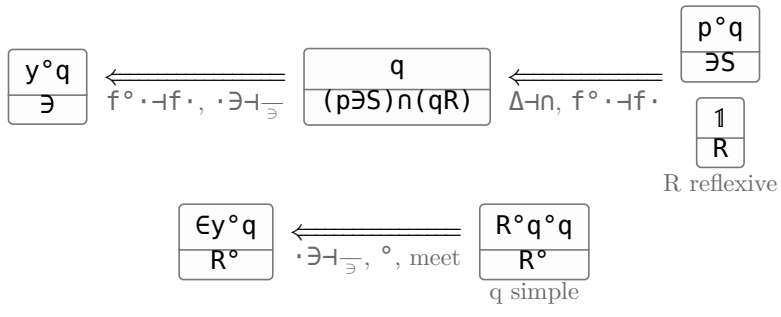
(11.1a)

(p,q) tabulates W≜(∋S)∩(∈\(SR°)), and y≜ $\frac{(p\ni S)\cap(qR)}{\ni}$, a map.

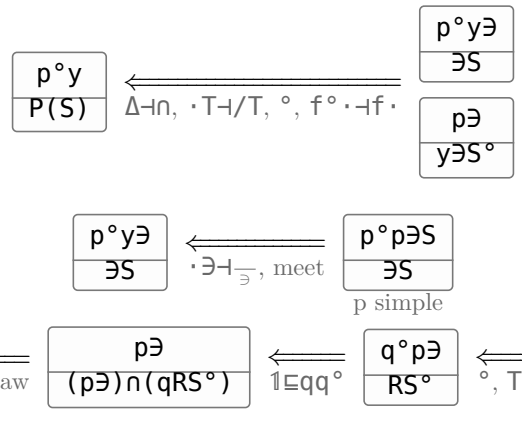
(11.1b)



(11.1c)

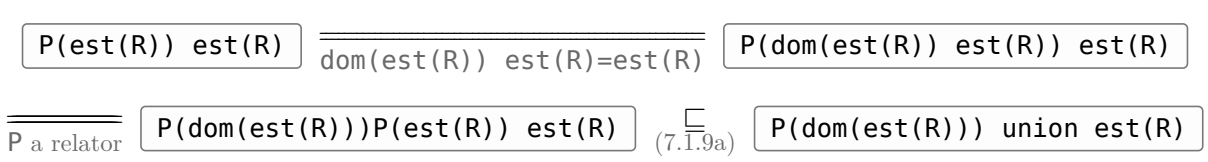


(11.1d)

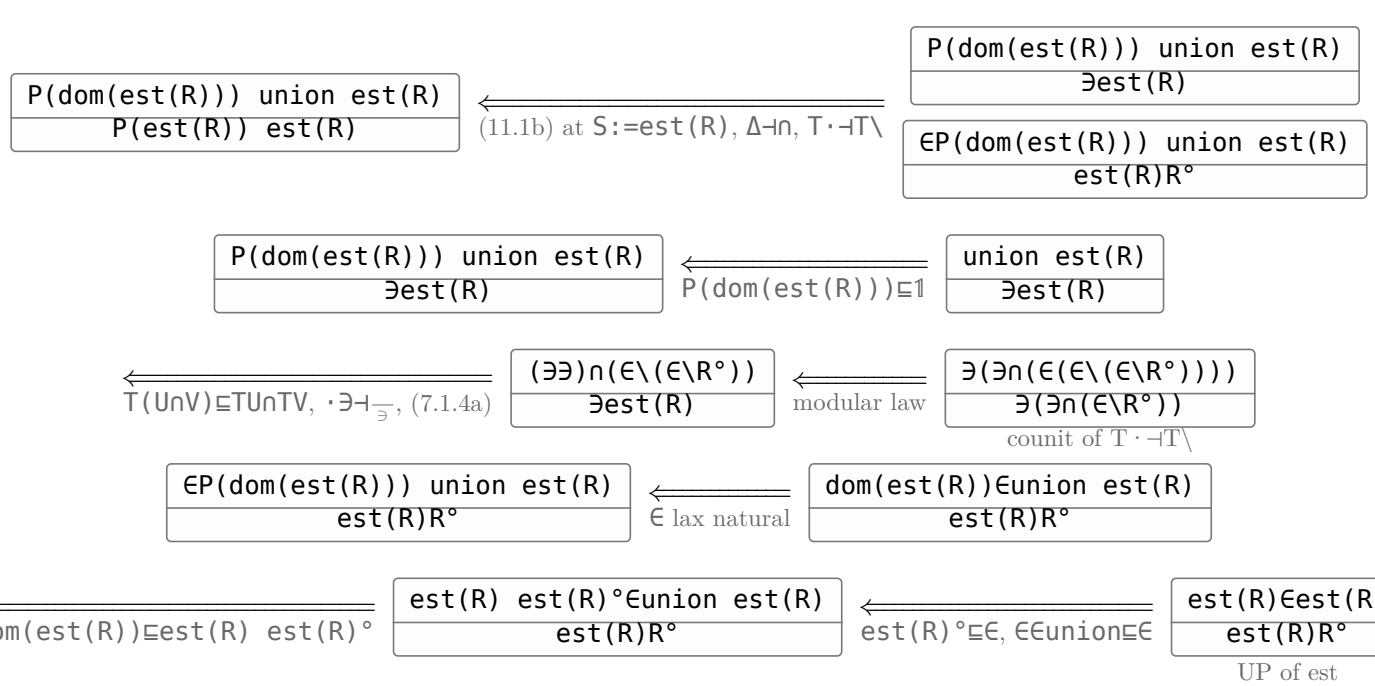


§11.2. P(est(R)) est(R)=P(dom(est(R))) union est(R)

(11.2a)



(11.2b)



§11.3. Shortest paths on a cylinder, on lists

(11.3a)

definition	**type**	**note**
`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.
`L=list⁺`	$\mathcal{A}\to\mathcal{A}$	A path is a non-empty list of squares, one per column crossed.
`α` the initial algebra	`F(dCL(L,E))→dCL(L,E)`	`α(5)=[5]` and `α(1,[5])=[1,5]`: a path is started by one square, or extended by one.
`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.
`R≜sum≤sum°`	`[Nat]→[Nat]`	The cost of a path, which the cheapest minimises.
`setify`	`A[n]→EA`	`setify(1,2,3,4)={1,2,3,4}` — which row a component came from is forgotten.
`moves`	`A[n]→E(A[n])`	`moves(x)={up(x),x,down(x)}` — rotated up, unrotated, rotated down.
`trans`	`E(A[n])→(E 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.
`zip`	`F(a[n])→(F a)[n]`	`zip((1,2,3,4),({[5]},{[6]},{[7]},{[8]}))=((1,{[5]}),(2,{[6]}),(3,{[7]}),(4,{[8]})).`
`cp≜` $\frac{\texttt{F(1,∋)}}{\texttt{∋}}$	`F(EA)→EFA`	`cp(1,{[5],[6],[8]})={(1,[5]),(1,[6]),(1,[8])}.`
`gen≜F(𝟙,moves trans N(union))zip N(cp(F(A,–),T)E(α))`	`F(NA,N(E(TA)))→N(E(TA))`	`gen((1,2,3,4),({[5]},{[6]},{[7]},{[8]}))` is worked out in (11.3.1b).
`paths≜⦇gen⦈setify union`	`T(NA)→E(TA)`	`paths[(1,2,3,4),(5,6,7,8)]` is the union of (11.3.1b)'s four sets: 12 paths, 3 from each entry row.
the specification `paths est(R)`	`[Nat[n]]→[Nat]`	A cheapest path from the entry side to the exit side.

§11.3.1. `N(E(L A))`

(11.3.1a)

```
1   5
2   6      [(1,2,3,4),(5,6,7,8)] : L N Nat
3   7
4   8
```

(11.3.1b)

```
⦇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]} )
```

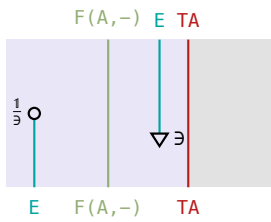
(11.3.1c)

N	the 4-tuple, one component per row of the entry column
E	the set of paths that can start in that row
L	a path is the non-empty list of the squares it crosses

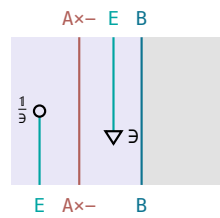
row 1's set has no `[1,7]` — from row 1 only rows 4, 1 and 2 are reachable — and it does have `[1,8]`, because the cylinder glues the bottom row to the top. That is `moves trans N(union)` of ≜ `paths`: component `k` collects the paths of rows `k-1`, `k`, `k+1`.

§11.3.2. The cross product cp≜$\frac{\texttt{F(1,∋)}}{\ni}$

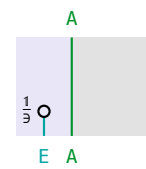
(11.3.2a)



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}`

(11.3.2b)

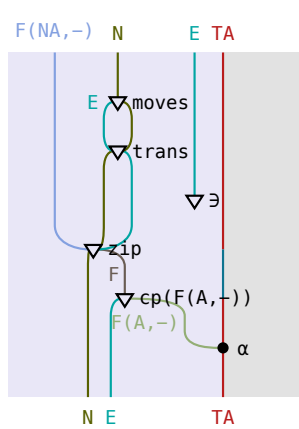
type	note
`F(A,−)=A+A×−`	one wire carries the whole functor, both summands with it; the `∋` under it is `F(𝟙,∋)`
`∋:E(L A)→L A`	the `∋` inside `F(𝟙,∋)`: a set of paths, one of them — `L A` is one path, `A` one square
`F(𝟙,∋)=𝟙+𝟙×∋`	`:A+A×E(L A)→A+A×L A`, the relator acting on each summand
`∋:E(A+A×L A)→A+A×L A`	the `∋` under the bar: a set of cells, one of them — a different `∋`
`cp:A+A×E(L A)→E(A+A×L A)`	$\frac{\texttt{R}}{\ni}$ turns `R:X→Y` into `X→E Y`
`cp=`$\frac{1}{\ni}$ `E(F(𝟙,∋))`	what the picture draws: the unit makes the outer `E`, leaving one `∋` — (3.1j)

(11.3.2c)

```
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
```

§11.3.3. gen=F(𝟙,moves trans N(union)) zip N(cp P(α))

(11.3.3a)

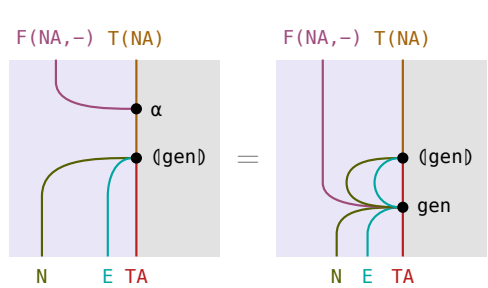


(11.3.3b)

```
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]
```

§11.3.4. gen is an F-algebra; ⦇gen⦈: α⦇gen⦈=F(𝟙,⦇gen⦈)gen

(11.3.4a)



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

(11.3.4b)

```
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 (11.3.1b): the tail folds to ({[5]},{[6]},{[7]},{[8]}), one path per row and each of them one square long, and gen on it is (11.3.3b)'s walk.

§11.3.5. 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)

(11.3.5a)

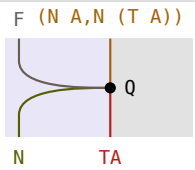
```
Q : F(N A,N(L A)) → N(L A)                    the fold's algebra: a new column, one path per row

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]
```

Q=[N(wrap),(𝟙×moves trans N(est(R))) zip' N(cons)] (11.3.5b)

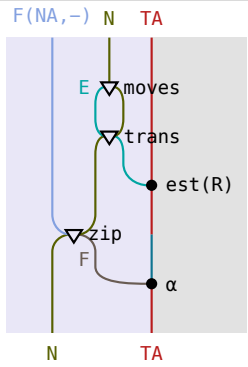
at the last column **Q** starts one path per row, and at each earlier one it puts each square in front of the cheapest of the three kept paths it can step to — the algebra (11.3.5c)'s last step folds

circuit — the fork is F(NA,N(LA))=NA+NA×N(LA) | **Hinze–Marsden**

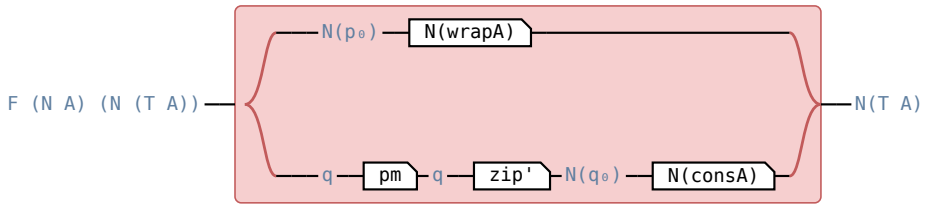


=

the fusion condition (11.3.5d) read as a definition,



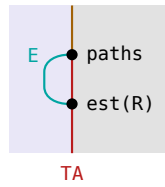
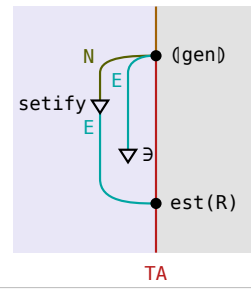
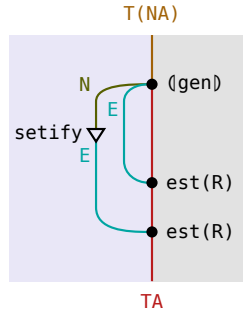
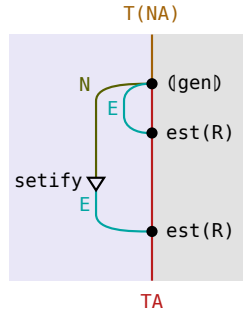
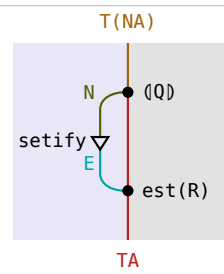
=



zip=𝟙+zip', α=[wrap,cons]

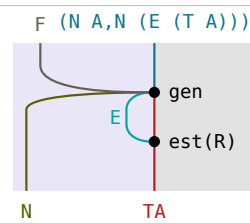
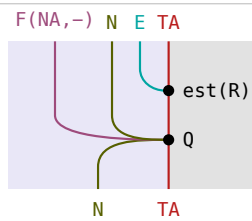
paths est(R)⊒⦇Q⦈ setify est(R) (11.3.5c)

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 (11.3.5b)'s algebra.

circuit	Hinze–Marsden
T (N A) — paths — E(T A) — est(R) — T A	
= T (N A) — [| F (N A) (N (E (T A))) — gen |] — NE(T A) — setify — EE(T A) — union — E(T A) — est(R) — T A ≜ paths at paths	
⊑ T (N A) — [| F (N A) (N (E (T A))) — gen |] — NE(T A) — setify — EE(T A) — P(est(R)) — E(T A) — est(R) — T A P(est(R)) est(R)⊑union est(R) — a minimum in each set, then a minimum of those; R transitive	
⊑ T (N A) — [| F (N A) (N (E (T A))) — gen |] — NE(T A) — N(est(R)) — N(T A) — setify — E(T A) — est(R) — T A setify lax natural	
⊑ T (N A) — [| F (N A) (N (T A)) — Q |] — N(T A) — setify — E(T A) — est(R) — T A fusion at (11.3.5d)	

gen N(est(R))⊒F(𝟙,N(est(R)))Q (11.3.5d)

fusion: the condition for fusion in (11.3.5c)'s last step, used to derive a definition of `Q`.

circuit	Hinze–Marsden
F (N A) (N (E (T A))) — gen — NE(T A) — N(est(R)) — N(T A)	
⊑ FNE(T A) — F(𝟙,N(est(R))) — FN(T A) — Q — N(T A) (7.13), then zip, trans, moves lax natural	

(7.13) is `F(𝟙,est(R))α⊑cp P(α) est(R)`, (7.2.1c) at the map `α` with $\frac{\texttt{F(𝟙,∋)α}}{\ni}$ `=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.