ALGORITHM DESIGN with HASKELL

Jeremy Gibbons
University of Oxford

Overview

1. *functional programming*

   equational reasoning for algorithm design

2. *greedy algorithms*

   simple examples of program calculation

3. *nondeterminism*

   a linguistic novelty

4. *thinning*

   an algorithmic novelty

1. Why functional programming matters

- programs are *values*, not commands
- supports *good old-fashioned equational reasoning*
- . . . with *program texts*, without needing a distinct language
- calculate *efficient implementations from clear specifications*

- using Haskell
- a plea for *simplicity*: no GADTs, no *Monad*s, no *Traversable*s. . .

Fusion

Consider the standard *foldr* function, eg $sum = foldr\ (+)\ 0$:

$$\begin{aligned} &foldr :: (\alpha \to \beta \to \beta) \to \beta \to [\,\alpha\,] \to \beta \\ &foldr\ f\ e\ [\,] \quad\quad = e \\ &foldr\ f\ e\ (x : xs) = f\ x\ (foldr\ f\ e\ xs) \end{aligned}$$

The *fusion law* for *foldr* states

$$h \cdot foldr\ f\ e = foldr\ f'\ e' \quad \Longleftarrow \quad \forall\, x\ y\ .\ h\ (f\ x\ y) = f'\ x\ (h\ y)\ \wedge\ h\ e = e'$$

For example,

$$double \cdot sum = foldr\ twicePlus\ 0 \quad \textbf{where}\ twicePlus\ x\ y = 2 \times x + y$$

because $double\ (x + y) = twicePlus\ x\ (double\ y)$ and $double\ 0 = 0$.

Fusion as the driving force

Many problems have a *simple specification* written with *standard components*:

$$algorithm = aggregate \cdot test \cdot generate$$

eg

$$mcst = minWith\ cost \cdot filter\ spanning \cdot trees$$

Aggregation is often *selection*. Generation of candidates often as a *fold* or *until*:

- *perms* (permutations)
- *segs* (segments)
- *parts* (partitions), *subseqs* (subsequences)
- *trees* (trees)

etc. Then the problem is to *fuse* the aggregation and testing with generation.

Questions? (1/4)

1. functional programming
2. greedy algorithms
3. nondeterminism
4. thinning

2. Greedy algorithms

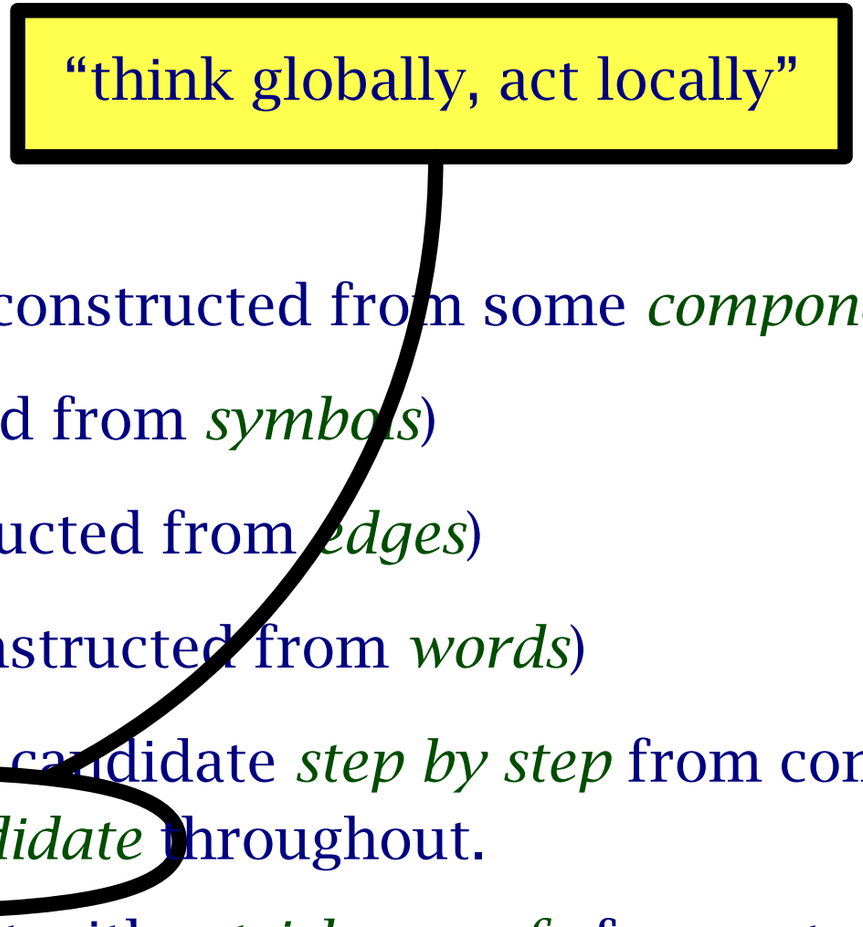
Selecting a single optimal *candidate* constructed from some *components*:

- eg shortest *encoding* (constructed from *symbols*)
- eg lightest spanning *tree* (constructed from *edges*)
- eg least wasteful *paragraph* (constructed from *words*)

Greedy algorithm assembles optimal candidate *step by step* from components, maintaining just a *single partial candidate* throughout.

Often leads to a *simple algorithm*, but with a *tricky proof* of correctness.

2. Greedy algorithms



Selecting a single optimal *candidate* constructed from some *components*:

- eg shortest *encoding* (constructed from *symbols*)
- eg lightest spanning *tree* (constructed from *edges*)
- eg least wasteful *paragraph* (constructed from *words*)

Greedy algorithm assembles optimal candidate *step by step* from components, maintaining just a *single partial candidate* throughout.

Often leads to a *simple algorithm*, but with a *tricky proof* of correctness.

A generic greedy algorithm

Computing a minimum-cost *Candidate* from some *Component*s:

$$mcc :: [\,Component\,] \to Candidate$$
$$mcc = minWith\ cost \cdot candidates$$

Here, *minWith* selects (leftmost) minimal element from a non-empty list:

$$minWith :: Ord\ b \Rightarrow (a \to b) \to [\,a\,] \to a$$
$$minWith\ f = foldr1\ (minBy\ f) \quad \textbf{where}\ minBy\ f\ x\ y = \textbf{if}\ f\ x \leqslant f\ y\ \textbf{then}\ x\ \textbf{else}\ y$$

and *candidates* constructs a finite non-empty list of *Candidate*s:

$$candidates :: [\,Component\,] \to [\,Candidate\,]$$
$$candidates = foldr\ step\ [\,c_0\,] \quad \textbf{where}\ step\ x\ cs = concat\ (map\ (extend\ x)\ cs)$$

where $c_0 :: Candidate$ and $extend :: Component \to Candidate \to [\,Candidate\,]$.

Greedy sorting

For *sorting strings*,

$$\textbf{type}\ \mathit{Candidate} = \mathit{String}$$
$$\textbf{type}\ \mathit{Component} = \mathit{Char}$$

The *empty* partial candidate is $c_0 =$ "", and candidates are *permutations*:

$$\mathit{extend}\ x\ c = [\, c_1 +\!\!+ [x] +\!\!+ c_2 \mid n \leftarrow [0\,..\,\mathit{length}\ c], \textbf{let}\ (c_1, c_2) = \mathit{splitAt}\ n\ c\,]$$

ie *extend* 'a' "bcd" = ["abcd","bacd","bcad","bcda"].

The cost to minimize is the *inversion count*:

$$\mathit{ic}\ xs = \mathit{length}\,[\,(x,y) \mid x : ys \leftarrow \mathit{tails}\ xs, y : zs \leftarrow \mathit{tails}\ ys, x > y\,]$$

Fusion

Recall *fusion law*

$$h \cdot foldr\ f\ e = foldr\ f'\ e' \quad \Longleftarrow \quad \forall x\ y\ .\ h\ (f\ x\ y) = f'\ x\ (h\ y)\ \wedge\ h\ e = e'$$

For greedy algorithm, $h = minWith\ cost$, $f = step$, $e = [\,c_0\,]$, $e' = c_0$,
and we have to find $f' = gstep$—the *greedy step*—such that:

$$minWith\ cost\ (step\ x\ cs) = gstep\ x\ (minWith\ cost\ cs)$$

Let us calculate!

Calculating *gstep*

$$
\begin{aligned}
& \mathit{minWith}\ \mathit{cost}\ (\mathit{step}\ x\ \mathit{cs}) \\
= \quad & [\![\ \text{definition of } \mathit{step}\ ]\!] \\
& \mathit{minWith}\ \mathit{cost}\ (\mathit{concat}\ (\mathit{map}\ (\mathit{extend}\ x)\ \mathit{cs})) \\
= \quad & [\![\ \text{distributive law}\ ]\!] \\
& \mathit{minWith}\ \mathit{cost}\ (\mathit{map}\ (\mathit{minWith}\ \mathit{cost} \cdot \mathit{extend}\ x)\ \mathit{cs}) \\
= \quad & [\![\ \text{define } \mathit{gstep}\ x = \mathit{minWith}\ \mathit{cost} \cdot \mathit{extend}\ x\ ]\!] \\
& \mathit{minWith}\ \mathit{cost}\ (\mathit{map}\ (\mathit{gstep}\ x)\ \mathit{cs}) \\
= \quad & [\![\ \text{greedy condition}\ ]\!] \\
& \mathit{gstep}\ x\ (\mathit{minWith}\ \mathit{cost}\ \mathit{cs})
\end{aligned}
$$

Calculating *gstep*

$$
\begin{aligned}
& \mathit{minWith}\ \mathit{cost}\ (\mathit{step}\ x\ \mathit{cs}) \\
= \quad & [\![\ \text{definition of } \mathit{step}\ ]\!] \\
& \mathit{minWith}\ \mathit{cost}\ (\mathit{concat}\ (\mathit{map}\ (\mathit{extend}\ x)\ \mathit{cs})) \\
= \quad & [\![\ \text{distributive law}\ ]\!] \\
& \mathit{minWith}\ \mathit{cost}\ (\mathit{map}\ (\mathit{minWith}\ \mathit{cost} \cdot \mathit{extend}\ x)\ \mathit{cs}) \\
= \quad & [\![\ \text{define } \mathit{gstep}\ x = \mathit{minWith}\ \mathit{cost} \cdot \mathit{extend}\ x\ ]\!] \\
& \mathit{minWith}\ \mathit{cost}\ (\mathit{map}\ (\mathit{gstep}\ x)\ \mathit{cs}) \\
= \quad & [\![\ \text{greedy condition}\ ]\!] \\
& \mathit{gstep}\ x\ (\mathit{minWith}\ \mathit{cost}\ \mathit{cs})
\end{aligned}
$$

$$\mathit{minWith}\ f\ (\mathit{concat}\ \mathit{xss}) = \mathit{minWith}\ f\ (\mathit{map}\ (\mathit{minWith}\ f)\ \mathit{xss})$$

Calculating *gstep*

minWith cost (step x cs)

minWith cost (concat (map (extend x) cs))

= [[distributive law]]

minWith cost (map (minWith cost · extend x) cs)

= [[define *gstep x* = *minWith cost · extend x*]]

minWith cost (map (gstep x) cs)

= [[greedy condition]]

gstep x (minWith cost cs)

minWith cost (map (gstep x) cs) = gstep x (minWith cost cs)

Calculating *gstep*

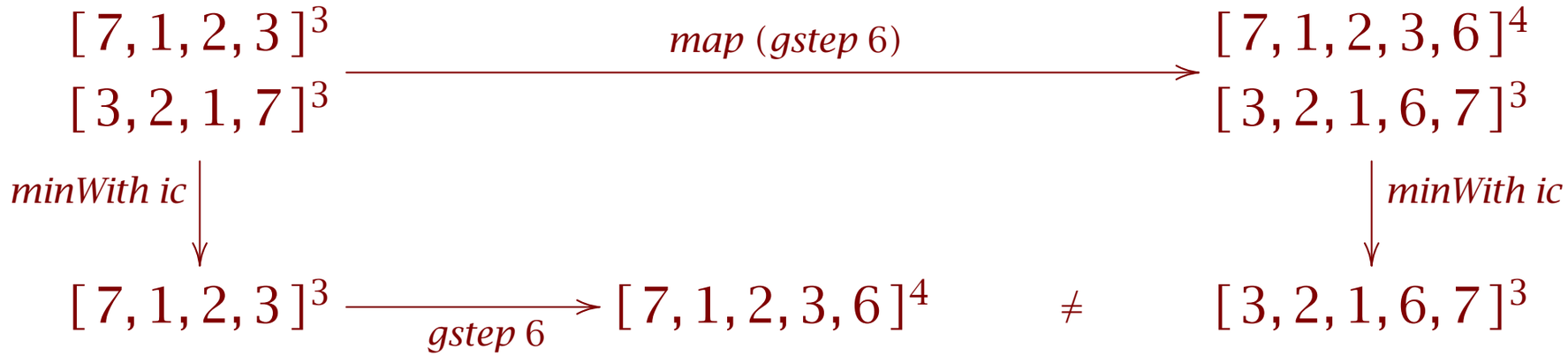
$$
\begin{aligned}
& \mathit{minWith}\ \mathit{cost}\ (\mathit{step}\ x\ \mathit{cs}) \\
= \quad & [\![\ \text{definition of } \mathit{step}\ ]\!] \\
& \mathit{minWith}\ \mathit{cost}\ (\mathit{concat}\ (\mathit{map}\ (\mathit{extend}\ x)\ \mathit{cs})) \\
= \quad & [\![\ \text{distributive law}\ ]\!] \\
& \mathit{minWith}\ \mathit{cost}\ (\mathit{map}\ (\mathit{minWith}\ \mathit{cost} \cdot \mathit{extend}\ x)\ \mathit{cs}) \\
= \quad & [\![\ \text{define } \mathit{gstep}\ x = \mathit{minWith}\ \mathit{cost} \cdot \mathit{extend}\ x\ ]\!] \\
& \mathit{minWith}\ \mathit{cost}\ (\mathit{map}\ (\mathit{gstep}\ x)\ \mathit{cs}) \\
= \quad & [\![\ \text{greedy condition}\ ]\!] \\
& \mathit{gstep}\ x\ (\mathit{minWith}\ \mathit{cost}\ \mathit{cs})
\end{aligned}
$$

So provided the greedy condition holds, we have the *greedy algorithm*:

$$\mathit{mcc} = \mathit{foldr}\ \mathit{gstep}\ c_0 \quad \textbf{where}\ \mathit{gstep}\ x = \mathit{minWith}\ \mathit{cost} \cdot \mathit{extend}\ x$$

Does greedy sorting work?

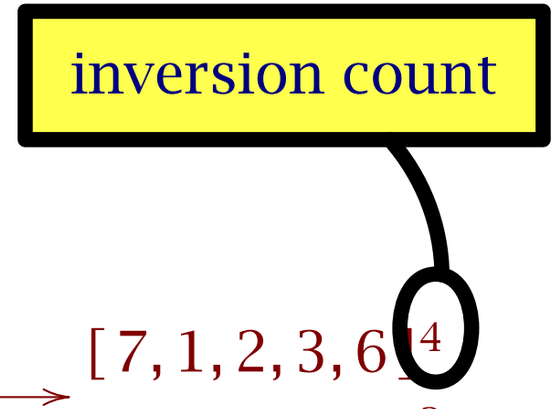
Sadly, no—the *greedy condition fails* for sorting:



In a nutshell, ordering by *ic* is *not linear*, because not antisymmetric. Fixes?

- *context-sensitive fusion* (unique optimum on range of *perms*)
- *refine the cost function* (from *ic* to *id*, which is a linear order)
- *nondeterminism* (keep all optimal candidates—more shortly)

Does greedy sorting work?



Sadly, no—the *greedy condition fails* for sorting:

$$
\begin{array}{ccc}
[\,7,1,2,3\,]^3 & \xrightarrow{\quad map\ (gstep\ 6)\quad} & [\,7,1,2,3,6\,]^4 \\
[\,3,2,1,7\,]^3 & & [\,3,2,1,6,7\,]^3 \\
\Big\downarrow{\scriptstyle minWith\ ic} & & \Big\downarrow{\scriptstyle minWith\ ic} \\
[\,7,1,2,3\,]^3 & \xrightarrow[gstep\ 6]{} [\,7,1,2,3,6\,]^4 & \neq \quad [\,3,2,1,6,7\,]^3
\end{array}
$$

In a nutshell, ordering by *ic* is *not linear*, because not antisymmetric. Fixes?

- *context-sensitive fusion* (unique optimum on range of *perms*)
- *refine the cost function* (from *ic* to *id*, which is a linear order)
- *nondeterminism* (keep all optimal candidates—more shortly)

Making change

UK coins  :

$$ukds = [1, 2, 5, 10, 20, 50, 100, 200] \quad \text{-- smallest assumed to be 1}$$

type *Tuple* = [*Nat*] is lists of counts, of same length as denominations:

$$count :: Tuple \to Nat$$
$$count = sum$$

Then fewest coins given by $change\ ds\ n = minWith\ count\ (tuples\ n\ ds)$, where

$$\begin{aligned}
&tuples\ n = finish \cdot foldr\ step\ [([\,], n)] \quad \textbf{where}\\
&\quad step\ d = concat \cdot map\ (extend\ d)\\
&\quad extend\ d\ (cs, r) = [(c : cs, r - c \times d) \mid c \leftarrow [0\,..\,r\ \underline{\text{div}}\ d]]\\
&\quad finish = map\ fst \cdot filter\ ((0 ==) \cdot snd)
\end{aligned}$$

Making change, greedily

Obvious greedy algorithm: *most coins possible, largest first*.

But greedy condition doesn't hold, because again the *ordering is not linear*:

—'obvious algorithm' chooses former, but *change* computes latter.
Desired implementation is not equal to specification.

Refine to a linear ordering: use *maxWith reverse* (ie lexically greatest, in reverse).

For UK decimal currency, greedy condition now holds; greedy algorithm emerges.

But for other currencies (eg UK pre-decimal), *greedy algorithm doesn't work*...

Questions? (2/4)

1. functional programming
2. greedy algorithms
3. nondeterminism
4. thinning

3. Nondeterminism

Refining to linear order is cheating: *ad hoc*, *prejudicial*, sometimes *impossible*.

But without it, establishing *greedy condition* for fusion

$$minWith\ cost\ (map\ (gstep\ x)\ cs) = gstep\ x\ (minWith\ cost\ cs)$$

entails proving the *equivalence*

$$cost\ c \leqslant cost\ c' \iff cost\ (gstep\ x\ c) \leqslant cost\ (gstep\ x\ c')$$

which rarely holds in practice. We need fusion to work given only *monotonicity*:

$$cost\ c \leqslant cost\ c' \implies cost\ (gstep\ x\ c) \leqslant cost\ (gstep\ x\ c')$$

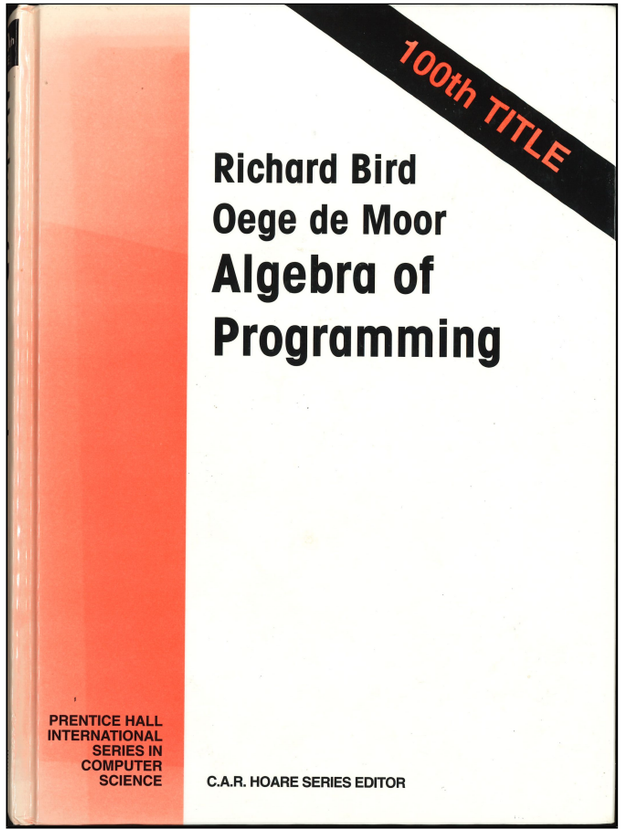
Relations

Optimization problems are often underdetermined:
multiple distinct minimal solutions.

Specification should admit *any optimal solution*,
not pre-commit to one.

But implementation must *choose one*:
should be optimal, but need not reach them all.

That is, implementation must *refine* specification.



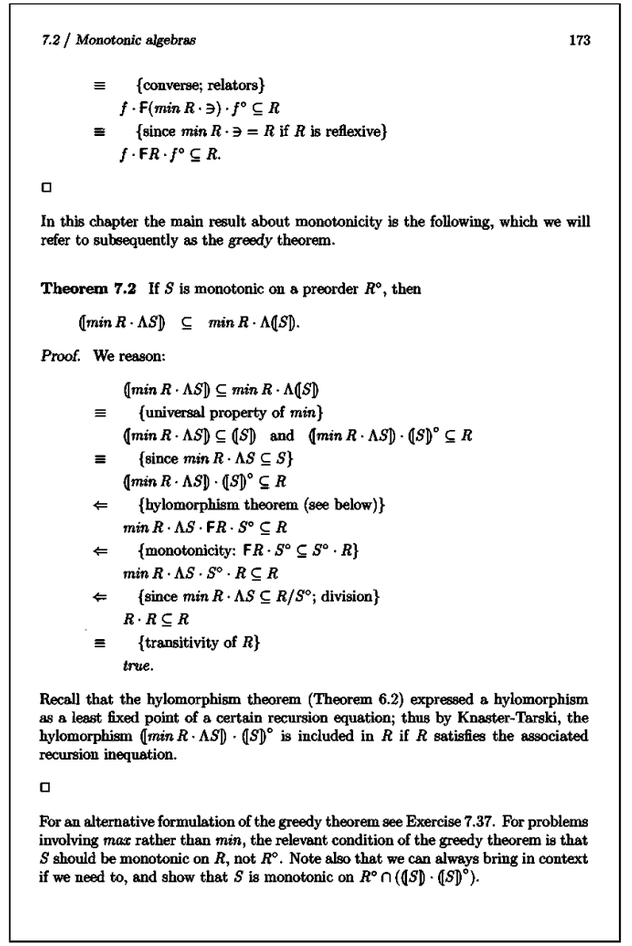
Algebra of Programming

Theorem 7.2 If S is monotonic on a preorder R°, then

$$(\!|\, min\, R \cdot \Lambda S \,|\!) \subseteq min\, R \cdot \Lambda(\!|\, S \,|\!)$$

Proof. We reason:

$$\begin{array}{ll}
& (\!|\, min\, R \cdot \Lambda S \,|\!) \subseteq min\, R \cdot \Lambda(\!|\, S \,|\!) \\
\equiv & \quad \{\text{universal property of } min\,\} \\
& (\!|\, min\, R \cdot \Lambda S \,|\!) \subseteq (\!|\, S \,|\!) \text{ and } (\!|\, min\, R \cdot \Lambda S \,|\!) \cdot (\!|\, S \,|\!)^\circ \subseteq R \\
\equiv & \quad \{\text{since } min\, R \cdot \Lambda S \subseteq S\,\} \\
& (\!|\, min\, R \cdot \Lambda S \,|\!) \cdot (\!|\, S \,|\!)^\circ \subseteq R \\
\Leftarrow & \quad \{\text{hylomorphism theorem}\,\} \\
& min\, R \cdot \Lambda S \cdot \mathsf{F}\, R \cdot S^\circ \subseteq R \\
\Leftarrow & \quad \{\text{monotonicity: } \mathsf{F}\, R \cdot S^\circ \subseteq S^\circ \cdot R\,\} \\
& min\, R \cdot \Lambda S \cdot S^\circ \cdot R \subseteq R
\end{array}$$



7.2 / Monotonic algebras 173

≡ {converse; relators}
$f \cdot \mathsf{F}(min\, R \cdot \ni) \cdot f^\circ \subseteq R$
≡ {since $min\, R \cdot \ni = R$ if R is reflexive}
$f \cdot \mathsf{F}R \cdot f^\circ \subseteq R$.

□

In this chapter the main result about monotonicity is the following, which we will refer to subsequently as the *greedy* theorem.

Theorem 7.2 If S is monotonic on a preorder R°, then

$(\!|\, min\, R \cdot \Lambda S \,|\!) \subseteq min\, R \cdot \Lambda(\!|\, S \,|\!)$.

Proof. We reason:

$(\!|\, min\, R \cdot \Lambda S \,|\!) \subseteq min\, R \cdot \Lambda(\!|\, S \,|\!)$
≡ {universal property of min}
$(\!|\, min\, R \cdot \Lambda S \,|\!) \subseteq (\!|\, S \,|\!)$ and $(\!|\, min\, R \cdot \Lambda S \,|\!) \cdot (\!|\, S \,|\!)^\circ \subseteq R$
≡ {since $min\, R \cdot \Lambda S \subseteq S$}
$(\!|\, min\, R \cdot \Lambda S \,|\!) \cdot (\!|\, S \,|\!)^\circ \subseteq R$
⇐ {hylomorphism theorem (see below)}
$min\, R \cdot \Lambda S \cdot \mathsf{F}R \cdot S^\circ \subseteq R$
⇐ {monotonicity: $\mathsf{F}R \cdot S^\circ \subseteq S^\circ \cdot R$}
$min\, R \cdot \Lambda S \cdot S^\circ \cdot R \subseteq R$
⇐ {since $min\, R \cdot \Lambda S \subseteq R/S^\circ$; division}
$R \cdot R \subseteq R$
≡ {transitivity of R}
true.

Recall that the hylomorphism theorem (Theorem 6.2) expressed a hylomorphism as a least fixed point of a certain recursion equation; thus by Knaster-Tarski, the hylomorphism $(\!|\, min\, R \cdot \Lambda S \,|\!) \cdot (\!|\, S \,|\!)^\circ$ is included in R if R satisfies the associated recursion inequation.

□

For an alternative formulation of the greedy theorem see Exercise 7.37. For problems involving *max* rather than *min*, the relevant condition of the greedy theorem is that S should be monotonic on R, not R°. Note also that we can always bring in context if we need to, and show that S is monotonic on $R^\circ \cap (\!|\, S \,|\!) \cdot (\!|\, S \,|\!)^\circ)$.

A gentler approach

It suffices to *extend* the development language slightly, with *MinWith* and $\leftsquigarrow$:

$$x \leftsquigarrow \mathit{MinWith}\ f\ xs \quad \Longleftrightarrow \quad x \in xs \ \wedge \ \forall y \in xs : f\ x \leqslant f\ y$$

For example, distributive law

$$\mathit{MinWith}\ \mathit{cost}\ (\mathit{concat}\ xss) = \mathit{MinWith}\ \mathit{cost}\ (\mathit{map}\ (\mathit{MinWith}\ \mathit{cost})\ xss)$$

means

$$\begin{aligned} & x \leftsquigarrow \mathit{MinWith}\ f\ (\mathit{concat}\ xss) \\ & \quad \Longleftrightarrow \quad x \leftsquigarrow \mathit{MinWith}\ f\ (\mathit{map}\ (\mathit{MinWith}\ f)\ xss) \\ & \quad \Longleftrightarrow \quad \exists xs\ .\ xs \leftsquigarrow \mathit{map}\ (\mathit{MinWith}\ f)\ xss \wedge x \leftsquigarrow \mathit{MinWith}\ f\ xs \end{aligned}$$

Just for *specifications*; no change to *implementation* language.

A gentler approach

It suffices to *extend* the development language slightly, with *MinWith* and $\leftsquigarrow$:

$$x \leftsquigarrow \mathit{MinWith}\ f\ xs \quad \Longleftrightarrow \quad x \in xs \ \wedge\ \forall y \in xs : f\ x \leqslant f\ y$$

For example, distributive law

$$\mathit{MinWith}\ \mathit{cost}\ (\mathit{concat}\ xss) = \mathit{MinWith}\ \mathit{cost}\ (\mathit{map}\ (\mathit{I}$$

means

$$\begin{aligned} & x \leftsquigarrow \mathit{MinWith}\ f\ (\mathit{concat}\ xss) \\ & \quad \Longleftrightarrow \quad x \leftsquigarrow \mathit{MinWith}\ f\ (\mathit{map}\ (\mathit{MinWith}\ f)\ xss) \\ & \quad \Longleftrightarrow \quad \exists xs\ .\ xs \leftsquigarrow \mathit{map}\ (\mathit{MinWith}\ f)\ xss \wedge x \leftsquigarrow \mathit{MinWith}\ f\ xs \end{aligned}$$

see Bird & Rabe,
How to Calculate with Nondeterministic Functions,
MPC 2019

Just for *specifications*; no change to *implementation* language.

Laws of nondeterministic functions

Fusion:

$$foldr\ f'\ e'\ xs \leadsto H\ (foldr\ f\ e\ xs) \quad \Longleftarrow \quad \forall x\ y\ .\ f'\ x\ (H\ y) \leadsto H\ (f\ x\ y)\ \wedge\ e' \leadsto H\ e$$

Then the greedy calculation goes through needing only *refinement*

$$gstep\ x\ (MinWith\ cost\ cs) \leadsto MinWith\ cost\ (map\ (gstep\ x)\ cs)$$

as a premise rather than equality, where

$$gstep\ x \leadsto MinWith\ cost \cdot extend\ x$$

Now nondeterministic *greedy condition holds* for changing coins, with $cost = count$.

Questions? (3/4)

1. functional programming
2. greedy algorithms
3. nondeterminism
4. thinning

4. Thinning

- greedy algorithm maintains a *single candidate*
- exhaustive search considers *all possible candidates*
- thinning is in between, maintaining *some candidates*
- some partial candidates *dominate* others, which may be discarded

- thinning not traditionally seen as a *distinct algorithm design technique*
- relevant problems often presented using *dynamic programming*
- the thinning presentation may lead to a *more effective solution*
- dynamic programming is better seen as a generalization of *divide & conquer*

Paths in a layered network

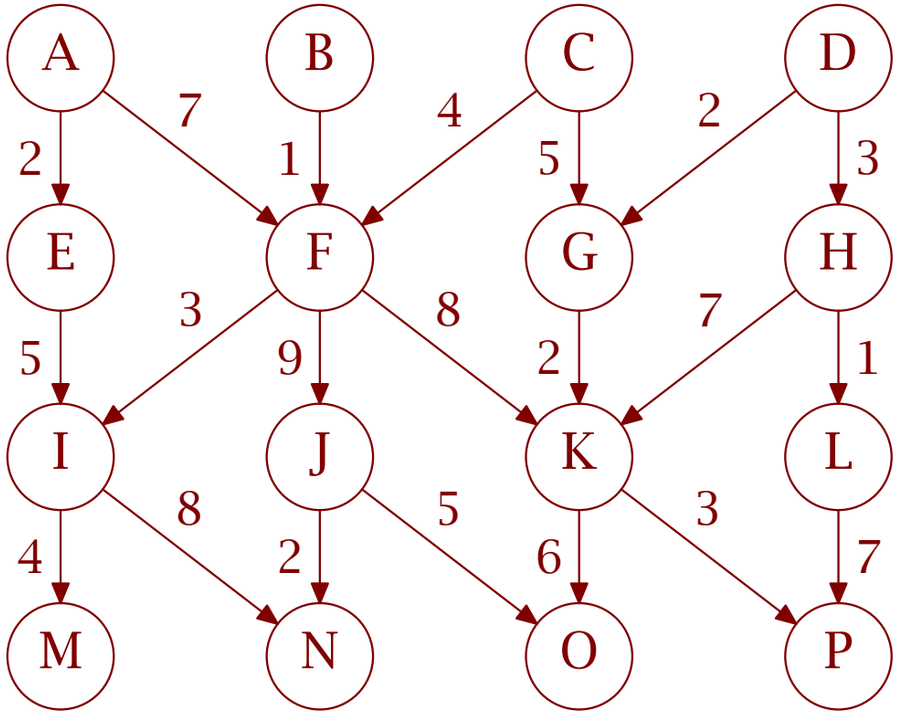
Directed, acyclic, layered, edge-weighted.

Find *shortest path* from top to bottom.

Dijkstra's algorithm takes $\Theta(n^2)$ steps.
We can do better.

Greedy algorithm doesn't work.

But from given source vertex,
a shorter path dominates a longer one.



Paths in a layered network

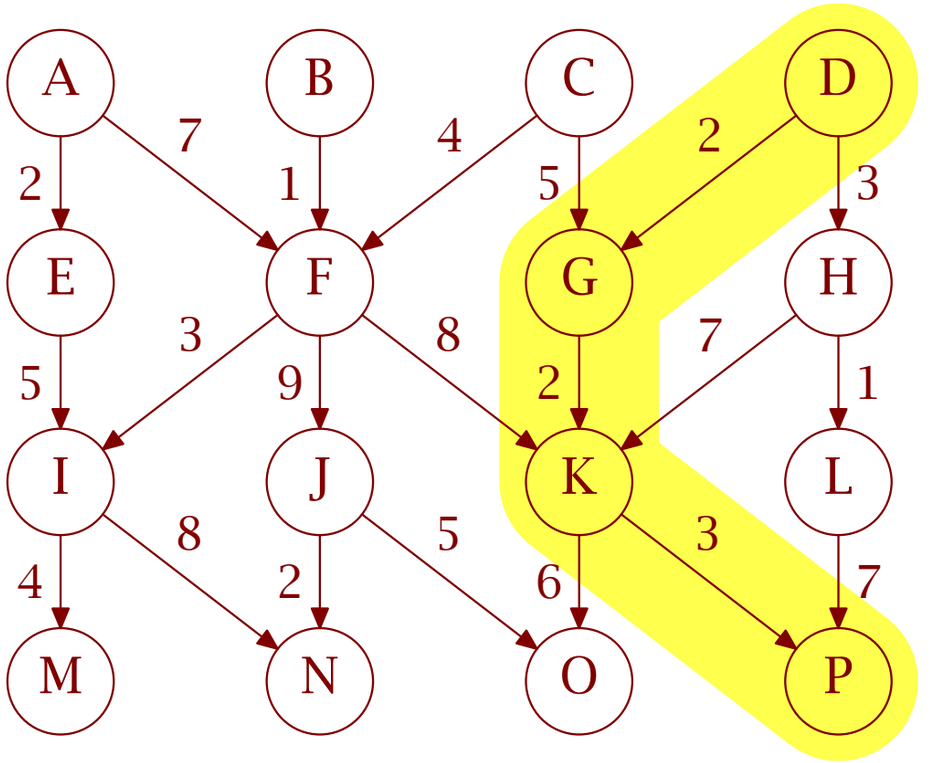
Directed, acyclic, layered, edge-weighted.

Find *shortest path* from top to bottom.

Dijkstra's algorithm takes $\Theta(n^2)$ steps.
We can do better.

Greedy algorithm doesn't work.

But from given source vertex,
a shorter path dominates a longer one.



Thinning

Another nondeterministic function

$$\mathit{ThinBy} :: (a \to a \to \mathit{Bool}) \to [a] \to [a]$$

returning some *dominating subsequence* under given preorder $\preceq$:

$$ys \leftsquigarrow \mathit{ThinBy}\ (\preceq)\ xs \quad \Longleftrightarrow \quad ys \sqsubseteq xs \wedge \forall x \in xs\ .\ \exists y \in ys\ .\ y \preceq x$$

Computing a *shortest thinning* takes quadratic time. Linear-time approximation:

$$\begin{aligned}
\mathit{thinBy}\ (\preceq) = \mathit{foldr}\ \mathit{bump}\ [\,]\quad \textbf{where}\ \mathit{bump}\ x\ [\,] &= [x] \\
\mathit{bump}\ x\ (y : ys) \mid x \preceq y &= x : ys \\
\mid y \preceq x &= y : ys \\
\mid \mathit{otherwise} &= x : y : ys
\end{aligned}$$

still useful if the candidates are generated in a convenient order.

Laws of thinning

$$MinWith\ cost = MinWith\ cost \cdot ThinBy\ (\preceq) \quad \Longleftarrow \quad x \preceq y \Rightarrow cost\ x \leqslant cost\ y$$

-- thin introduction

$$wrap \cdot MinWith\ cost \leftsquigarrow ThinBy\ (\preceq) \quad \Longleftarrow \quad cost\ x \leqslant cost\ y \Rightarrow x \preceq y$$

-- thin elimination

$$ThinBy\ (\preceq) \cdot concat = ThinBy\ (\preceq) \cdot concatMap\ (ThinBy\ (\preceq))$$

-- distributivity

$$map\ f \cdot ThinBy\ (\preceq) \leftsquigarrow ThinBy\ (\preceq) \cdot map\ f \quad \Longleftarrow \quad x \preceq y \Rightarrow f\ x \preceq f\ y$$

-- thin–map

$$ThinBy\ (\preceq) \cdot map\ f \leftsquigarrow map\ f \cdot ThinBy\ (\preceq) \quad \Longleftarrow \quad f\ x \preceq f\ y \Rightarrow x \preceq y$$

-- thin–map

$$ThinBy\ (\preceq) \cdot filter\ p = filter\ p \cdot ThinBy\ (\preceq) \quad \Longleftarrow \quad (x \preceq y \wedge p\ y) \Rightarrow p\ x$$

-- thin–filter

Specifying the problem

```
data Edge    = E {source :: Vertex, target :: Vertex, weight :: Weight}
mcp :: [[Edge]] → [Edge]                -- network to path
mcp ⇜ MinWith cost · paths

cost :: [Edge] → Weight
cost = sum · map weight

paths :: [[Edge]] → [[Edge]]            -- network to set of paths
paths = foldr step [[]]  where
   step es ps = concat [cons e ps | e ← es]
   cons e ps  = [e : p | p ← ps, linked e p]
   linked e p = null p ∨ (target e == source (head p))
```

Introducing thinning

Thin introduction lets us *refine* the specification to

$$mcp \leftsquigarrow MinWith\ cost \cdot ThinBy\ (\preceq) \cdot paths$$

provided that $p \preceq q \Rightarrow cost\ p \leqslant cost\ q$. Choose

$$p \preceq q \quad = \quad (source\ (head\ p) == source\ (head\ q)) \wedge (cost\ p \leqslant cost\ q)$$

Aim now to fuse $ThinBy\ (\preceq)$ and $paths$: find $tstep$ satisfying *fusion condition*

$$tstep\ es\ (ThinBy\ (\preceq)\ ps) \leftsquigarrow ThinBy\ (\preceq)\ (step\ es\ ps)$$

Let us calculate!

Calculating the thinning step

$$
\begin{aligned}
& \mathit{ThinBy}\ (\preceq)\ (\mathit{step}\ es\ ps) \\
= \quad & [\![\ \text{definition of } \mathit{step}\ ]\!] \\
& \mathit{ThinBy}\ (\preceq)\ (\mathit{concat}\ [\mathit{cons}\ e\ ps \mid e \leftarrow es]) \\
= \quad & [\![\ \text{distributivity}\ ]\!] \\
& \mathit{ThinBy}\ (\preceq)\ (\mathit{concat}\ [\mathit{ThinBy}\ (\preceq)\ (\mathit{cons}\ e\ ps) \mid e \leftarrow es]) \\
= \quad & [\![\ \text{claim (see book)}\ ]\!] \\
& \mathit{ThinBy}\ (\preceq)\ (\mathit{concat}\ [\mathit{cons}\ e\ (\mathit{ThinBy}\ (\preceq)\ ps) \mid e \leftarrow es]) \\
= \quad & [\![\ \text{definition of } \mathit{step}\ ]\!] \\
& \mathit{ThinBy}\ (\preceq)\ (\mathit{step}\ es\ (\mathit{ThinBy}\ (\preceq)\ ps)) \\
\rightsquigarrow \quad & [\![\ \text{defining } \mathit{tstep}\ es\ ps \rightsquigarrow \mathit{ThinBy}\ (\preceq)\ (\mathit{step}\ es\ ps)\ ]\!] \\
& \mathit{tstep}\ es\ (\mathit{ThinBy}\ (\preceq)\ ps)
\end{aligned}
$$

The resulting algorithm

We have calculated

$$
\begin{aligned}
&\mathit{mcp} = \mathit{minWith}\ \mathit{cost} \cdot \mathit{foldr}\ \mathit{tstep}\ [[\,]] \quad \textbf{where}\\
&\quad \mathit{tstep}\ \mathit{es}\ \mathit{ps} = \mathit{thinBy}\ (\preceq)\ (\mathit{step}\ \mathit{es}\ \mathit{ps})\\
&\quad \mathit{step}\ \mathit{es}\ \mathit{ps}\ \ = [\,e : p \mid e \leftarrow \mathit{es}, p \leftarrow \mathit{ps}, \mathit{linked}\ e\ p\,]
\end{aligned}
$$

where *minWith* and *thinBy* implement *MinWith* and *ThinBy*.

Optimization: *tuple* paths with their costs, to avoid recomputation.

Optimization: *sort* layers and candidates by source vertex: ≤ 1 path per vertex.

With d layers of k vertices, thinning takes $O(d^3\,k)$, whereas Dijkstra's algorithm takes $O(d^2\,k^2)$, so better when $d > k$. Using *arrays*, comes down to $O(d^2\,k)$.

Thinning coins

Greedy algorithm does *not* work for all currencies, eg 48d in UK pre-decimal:

 ($^1/_2$d),  (1d),  (3d),  (6d),  (12d),  (24d),  (30d).

But *thinning works*. Partial candidates (cs, r) with counts cs and residue r, and

$$cost\ (cs, r) = (r, count\ cs)$$

Thin by some $\preceq$ such that $x \preceq y \Rightarrow cost\ x \leqslant cost\ y$; choose

$$(cs, r) \preceq (cs', r') \quad = \quad (r == r') \wedge (count\ cs \leqslant count\ cs')$$

—of two candidates with *same residue, smaller count* dominates larger.

(It is also a *layered network problem*. And susceptible to *dynamic programming*.)

Summary

1. *functional programming*

   equational reasoning for algorithm design

2. *greedy algorithms*

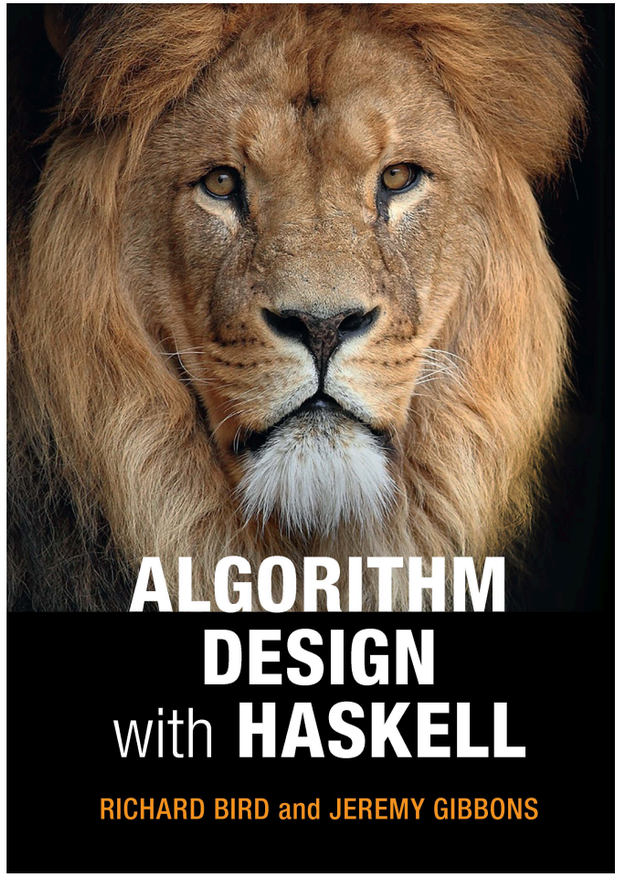
   simple examples of program calculation

3. *nondeterminism*

   a linguistic novelty

4. *thinning*

   an algorithmic novelty



CUP, out now!

Questions? (4/4)

@jer_gib

jeremy.gibbons@cs.ox.ac.uk

www.cambridge.org/core/books/algorithm-design-with-haskell/824BE0319E3762CE8BA5B1D91EEA3F52

£ drop me a line for 20% discount voucher