Chapter 8
Algebraic Methods for Optimization Problems

Richard Bird, Jeremy Gibbons and Shin-Cheng Mu

Abstract. We argue for the benefits of relations over functions for modelling programs, and even more so for modelling specifications. To support this argument, we present an extended case study for a class of optimization problems, deriving efficient functional programs from concise relational specifications.

1 Introduction

It is very natural to model computer programs as functions from input to output; hence our interest in functional programming, and the approaches to program calculation described in Chapter 5.

However, sometimes the functional view is too concrete, too implementation-oriented. Although functions may be a good model of *programs*, they are not a particularly good model of *specifications*. It is often more natural to specify a problem using features beyond the power of functional programming. Here are some examples.

- It is easy to write the squaring problem *sqr* as a function. A natural specification of the square root problem *sqrt* is as its *converse*: the result of *sqrt* x should be a y such that *sqr* $y = x$ (subject to considerations of sign and rounding). A more elaborate application is to specify a parser as the converse of a pretty printer.
- Many problems involve computing a minimal — smallest, cheapest, best — element of a set under some ordering — size, cost, badness. (We will see some examples of such *optimization problems* later in this chapter.) Often the ordering is a preorder rather than a partial order: two different elements may be equivalent under the ordering. Then the problem *minimal* r, which computes a minimal element under ordering r, is *non-deterministic*.
- The problem *minimal* r cannot return any result on an empty set. Even on a non-empty set, for there always to be a minimal element the ordering must be *connected* (any two elements must be comparable, one way or the other); for an unconnected ordering there may again be no returnable result. So *minimal* r will in general also be *partial*.
- Many problems involve simultaneously satisfying two requirements P and Q; for example, sorting a sequence x under an ordering r involves constructing a permutation (requirement P) of x that is ordered according to r (requirement Q). One can specify such a problem by sequentially composing two

subproblems: first find all elements satisfying requirement P, then discard those that do not satisfy requirement Q. However, doing so inevitably breaks the symmetry in the problem description, thereby introducing a bias towards one kind of solution. A more neutral specification retains the symmetry by simply forming the *conjunction* of the two subproblems.

- Dually, some problems involve satisfying one of two alternative requirements — perhaps two different ways of giving an answer, or perhaps a 'normal' case and an 'exceptional' or 'error' case. Each subproblem is partial, but the two together may be total. Modelling the problem as a function entails entangling the two aspects; a better separation of concerns is obtained by modelling each aspect independently and forming the *disjunction* of the two subproblems.

In summary, many problems are most naturally *specified* using features (converse, non-determinism, partiality, conjunction, disjunction) that are to varying degrees inappropriate for a *programming* language.

The obvious way of resolving this dilemma is to use two languages, one for specification and one for programming. In fact, it is most convenient if the programming language is a sublanguage of the specification language. Then program construction is a matter of reasoning within a single language, namely the wider specification language, and restricting the features used to obtain an acceptable expression in the implementable subset. Moreover, it is then straightforward to have intermediate expressions that are a mixture of program and specification; this is rather harder to achieve if the two are completely separate languages.

What could we choose as a language that encompasses functions and functional programming, but that also provides the expressive power of converse, non-determinism, conjunction, and so on? It is well-known that *relations* have exactly these properties. This chapter, therefore, looks at lifting the results of Chapter 5 from functions to relations. We will carry out an extended case study, illustrating the gain in expressivity provided by the extra features. Our programming methodology will be to *refine* a relational specification to a functional implementation, reducing non-determinism while maintaining the domain of definition. So we will take this case study all the way to code, as a functional program.

There is an added bonus from studying relations as well as functions. Even when a specification and its implementation are both functional, one sometimes can gain greater insight and simpler reasoning by carrying out the calculation of the latter from the former in more general theory of relations. (We see an example in Exercise 2.6.7.) This is analogous to complex numbers and trigonometry in high school mathematics: although trigonometric functions are completely 'real' concepts, it is enlightening to investigate them from the point of view of complex numbers. For example, $\sin\theta$ and $\cos\theta$ are the imaginary and real parts of $e^{\theta i}$, the complex number with 'angle' θ radians and unit 'radius'; so of course $\sin^2\theta + \cos^2\theta = 1$, by Pythagoras' Theorem, and the 'most beautiful theorem of mathematics' $e^{\pi i} = -1$ is easy to visualize.

It turns out that the relational algebra is particularly powerful for specifying and reasoning about optimization problems — informally, problems of the form 'the best data structure constructed in this way, satisfying that condition'. This chapter is structured to lead to exactly this point: Section 2 introduces the algebra of relations, Section 3 develops some general-purpose theory for optimization problems, and Section 4 applies the theory in a case study, which we take all the way to runnable code. There are many interesting diversions en route, some of which are signposted in the exercises; more sources are provided in the bibliographic notes.

1.1 Bibliographic notes

The relational algebra has been proposed as a basis for program construction by many people; some representative strands of work are by de Roever [10], Hoare [17], Berghammer [3], Möller [24], Backhouse [1, 2], Hehner [16], Mili and Desharnais [23], and Bird and de Moor [5]. Indeed, one could argue that relations underlie the entire field of logic programming [22, 29].

For more information about the use of relations for program calculation, particularly in the solution of optimization problems, the interested reader is directed towards [5], on which this chapter is based.

2 The algebra of relations

2.1 Relations

Like a function, a *relation* is a mapping between a source type and a target type; unlike a function, a relation may be *partial* (mapping a source value to no target value) and/or *non-deterministic* (mapping a source value to multiple target values). We write '$f :: \mathsf{A} \leadsto \mathsf{B}$' to denote that relation f is a mapping from source type A to target type B.

2.1.1 Example

For example, consider the datatype $\mathsf{PList\ A}$ of non-empty lists with elements drawn from A. In general, lists xs of this type can be split into a prefix ys and a suffix zs such that $ys +\!\!+ zs = xs$ (here, '$+\!\!+$' denotes list concatenation):

$$split :: \mathsf{PList\ A} \leadsto \mathsf{PList\ A} \times \mathsf{PList\ A}$$

Most lists can be split in many ways, corresponding to the choice of where to place the division, so *split* is non-deterministic; for example, both $([1, 2], [3])$ and $([1], [2, 3])$ are possible results of $split\ [1, 2, 3]$. Singleton lists can not be split in any way at all, so *split* is partial too.

2.1.2 Pointwise relational programming

Non-deterministic languages discourage reasoning with variables, because of the care that needs to be taken to retain referential transparency. Basically, one cannot assign to a variable 'the result of running program p on input x', because there may be no or many results. The usual approach, then, is to eschew *pointwise* reasoning with variables representing values, in favour of *pointfree* reasoning with the programs themselves.

Nevertheless, there are times when it is much more convenient to name and reason about particular values. This is especially the case when working on applications rather than general-purpose theory, because one wants to name and capitalize on specific dataflow patterns. The right approach to take to this problem is still not clear; the approach we will follow in this chapter is just one of several, and the details are not yet completely worked out. To be on the safe side, we will not use these 'pointwise relational programming' notations for any formal purpose; rather, we will resort to them only to provide informal but more perspicuous characterizations of sometimes opaque relational expressions.

When we want to refer to the result of 'applying' a relational program, we will avoid using '=', and write instead '$y \mathrel{\leftsquigarrow} f\ x$' to mean that y is a possible value of $f\ x$. For example, both

$$([1,2],[3]) \mathrel{\leftsquigarrow} \mathit{split}\ [1,2,3]$$

and

$$([1],[2,3]) \mathrel{\leftsquigarrow} \mathit{split}\ [1,2,3]$$

In functional programming, we find that it is often convenient to specify a function by *pattern matching*. The function is defined by one or more equations, but the arguments to the function on the left-hand side of the equations are *patterns* rather than simply variables. For example, we might define the factorial function on natural numbers by

$$\begin{aligned} \mathit{fact}\ 0 &= 1 \\ \mathit{fact}\ (n+1) &= (n+1) \times \mathit{fact}\ n \end{aligned}$$

This works for functions because patterns are by definition injective (and usually non-overlapping), and not arbitrary expressions: for any natural argument, exactly one of these two equations applies, and moreover if it is the second equation, the value of n is uniquely determined. Definition by pattern matching is often just as convenient for relations, but with relations we can be more lenient: we can allow non-exhaustive equations (which will lead to partiality) and overlapping equations with non-injective patterns (which will lead to non-determinism). For example, we could define the function *split* above by

$$\mathit{split}\ (ys +\!\!+ zs) \mathrel{\widehat{=}} (ys, zs)$$

This pattern is not injective; nevertheless, this equation — together with the type — completely determines the relation *split*. (We decorate the equals sign to emphasize that this is the definition of a relation, rather than a true identity of values; in particular, $=$ is transitive whereas $\widehat{=}$ is not.)

Non-injective patterns introduce non-determinism implicitly. Explicit non-determinism can be expressed using the choice operator $\Box$. For example, here is a more constructive characterization of *split*:

$$\begin{aligned} &\mathit{split}\ (\mathit{consp}\ (x, xs)) = ([x], xs) \Box (\mathit{consp}\ (x, ys), zs) \\ &\qquad\qquad\qquad\qquad \text{where } (ys, zs) \leftsquigarrow \mathit{split}\ xs \end{aligned}$$

Here, '*consp*' denotes the prefixing of a single element onto a non-empty list. The pattern is injective — it matches in at most one way for any given list — but still the result is non-deterministic because of the explicit choice.

2.1.3 Composition

Relations of appropriate types may be composed; if $f :: \mathsf{A} \rightsquigarrow \mathsf{B}$ and $g :: \mathsf{B} \rightsquigarrow \mathsf{C}$, then their composition $g \cdot f :: \mathsf{A} \rightsquigarrow \mathsf{C}$ is defined by

$$z \leftsquigarrow (g \cdot f)\ x \ \Leftrightarrow\ \exists y.\ y \leftsquigarrow f\ x \wedge z \leftsquigarrow g\ y$$

For every type A, there is an *identity* relation $\mathsf{id}_\mathsf{A} :: \mathsf{A} \rightsquigarrow \mathsf{A}$ mapping each element of A to itself. Identities are the units of composition: for $f :: \mathsf{A} \rightsquigarrow \mathsf{B}$,

$$f \cdot \mathsf{id}_\mathsf{A} = f = \mathsf{id}_\mathsf{B} \cdot f$$

We will usually omit the subscript, allowing it to be deduced from the context.

2.1.4 Inclusion

One can think of a relation $f :: \mathsf{A} \rightsquigarrow \mathsf{B}$ as a set of pairs, a subset of $\mathsf{A} \times \mathsf{B}$ (so $y \leftsquigarrow f\ x$ precisely when $(x, y) \in f$). Unlike functions, different relations of the same type may be comparable under *inclusion*: $f \subseteq g$ precisely when $y \leftsquigarrow f\ x$ implies $y \leftsquigarrow g\ x$ for all x and y.

Composition is monotonic under inclusion:

$$f_1 \subseteq f_2\ \wedge\ g_1 \subseteq g_2\ \Rightarrow\ f_1 \cdot g_1 \subseteq f_2 \cdot g_2$$

Moreover, pre- and post-composition have adjoints, called *right division* ('over') and *left division* ('under') respectively:

$$(f\ \subseteq\ h\ /\ g)\ \Leftrightarrow\ (f \cdot g\ \subseteq\ h)\ \Leftrightarrow\ (g\ \subseteq\ f \setminus h)$$

With points,

$$\begin{aligned} &a \leftsquigarrow (h\ /\ g)\ b\ \Leftrightarrow\ \forall c.\ (a \leftsquigarrow h\ c \Leftarrow b \leftsquigarrow g\ c) \\ &a \leftsquigarrow (f \setminus h)\ c\ \Leftrightarrow\ \forall b.\ (b \leftsquigarrow f\ a \Rightarrow b \leftsquigarrow h\ c) \end{aligned}$$

Inclusion of relations is the foundation of our programming method, as equality is for functional programming: we will start with a (presumably non-deterministic) specification g, and manipulate it to construct a (presumably more efficient, or otherwise easier to implement) refinement $f \subseteq g$. Provided that f is still defined everywhere that g was, it is 'at least as good' as g.

2.1.5 Meet and join

Any two relations $f, g :: \mathsf{A} \rightsquigarrow \mathsf{B}$ have a *meet* $f \cap g :: \mathsf{A} \rightsquigarrow \mathsf{B}$, defined by the universal property that for all $h :: \mathsf{A} \rightsquigarrow \mathsf{B}$,

$$h \subseteq f \cap g \;\Leftrightarrow\; h \subseteq f \wedge h \subseteq g$$

That is, $f \cap g$ is the greatest lower bound of f and g; it corresponds to intersection of the sets of pairs in the relations. It follows (Exercise 2.6.1) that $\cap$ is associative, commutative and idempotent, and that monotonicity of composition can be re-expressed in terms of $\cap$.

Dually, the join $f \cup g$ of two relations f, g of the same type is defined by the universal property

$$h \supseteq f \cup g \;\Leftrightarrow\; h \supseteq f \wedge h \supseteq g$$

2.1.6 Converse

Again unlike functions, relations can easily be reversed: if $f :: \mathsf{A} \rightsquigarrow \mathsf{B}$, then $f^\circ :: \mathsf{B} \rightsquigarrow \mathsf{A}$, and satisfies

$$y \mathrel{\leftsquigarrow} f\ x \;\Leftrightarrow\; x \mathrel{\leftsquigarrow} f^\circ\ y$$

Converse is:

- its own inverse: $(f^\circ)^\circ = f$;
- order-preserving: $f \subseteq g \Leftrightarrow f^\circ \subseteq g^\circ$;
- contravariant: $(f \cdot g)^\circ = g^\circ \cdot f^\circ$;
- distributive over meet: $(f \cap g)^\circ = f^\circ \cap g^\circ$.

Any *coreflexive* f, that is, one satisfying $f \subseteq \mathsf{id}$, is invariant under converse: $f^\circ = f$. One can think of a coreflexive of type $\mathsf{A} \rightsquigarrow \mathsf{A}$ as a subset of A, or equivalently as a predicate on A.

2.2 Special kinds of relation

Let $f :: \mathsf{A} \rightsquigarrow \mathsf{B}$; then

- f is *entire* if $\mathsf{id} \subseteq f^\circ \cdot f$, or equivalently if for all $x \in \mathsf{A}$ there is at least one $y \in \mathsf{B}$ with $y \mathrel{\leftsquigarrow} f\ x$;
- f is *simple* if $f \cdot f^\circ \subseteq \mathsf{id}$, or equivalently if for all $x \in \mathsf{A}$ there is at most one $y \in \mathsf{B}$ with $y \mathrel{\leftsquigarrow} f\ x$;
- f is *surjective* if $\mathsf{id} \subseteq f \cdot f^\circ$, or equivalently if for all $y \in \mathsf{B}$ there is at least one $x \in \mathsf{A}$ with $y \mathrel{\leftsquigarrow} f\ x$;
- f is *injective* if $f^\circ \cdot f \subseteq \mathsf{id}$, or equivalently if for all $y \in \mathsf{B}$ there is at most one $x \in \mathsf{A}$ with $y \mathrel{\leftsquigarrow} f\ x$.

A simple relation is also known as a *partial function*, and a simple and entire relation as a *total function*. We write '$f :: \mathsf{A} \to \mathsf{B}$' to indicate that f is a total function. Functions enjoy special *shunting rules*:

Lemma 1. *If f is a function, then*

$$f \cdot g \subseteq h \;\Leftrightarrow g \subseteq f^\circ \cdot h$$
$$g \cdot f^\circ \subseteq h \Leftrightarrow g \subseteq h \cdot f$$

Note that $f \subseteq g \Rightarrow f = g$ if f, g are total functions. For function $p :: \mathsf{A} \to \mathsf{Bool}$, we define the corresponding coreflexive $p? :: \mathsf{A} \rightsquigarrow \mathsf{A}$ by

$$p? = \mathsf{exr} \cdot \mathit{fsttrue} \cdot (p \vartriangle \mathsf{id})$$

where the coreflexive *fsttrue* holds of pairs whose first component is true.

2.3 Breadth

There is a one-to-one correspondence between relations $\mathsf{A} \rightsquigarrow \mathsf{B}$ and set-valued functions $\mathsf{A} \to \mathsf{Set}\ \mathsf{B}$, where Set is the powerset functor ($\mathsf{Set}\ \mathsf{B}$ is the set of all subsets of B). The operator Λ is a witness to this isomorphism, yielding the *breadth* of a relation, that is, the corresponding set-valued function:

$$(\Lambda f)\ x = \{y \mid y \mathrel{\leftsquigarrow} f\ x\}$$

For example, $\Lambda \mathit{split}$ is the function that returns the set of all possible splits of a given list.

Note that Λf is an entire and simple relation (a function) for any f. If f itself is a function, then Λf returns singleton sets.

2.4 Folds

Functional folds were discussed in depth in Chapter 5. To summarize, an F-*algebra* (A, f) for a functor F consists of a type A and a function $f :: \mathsf{F}\ \mathsf{A} \to \mathsf{A}$. An F-algebra $(\mathsf{T}, \mathsf{in}_\mathsf{T})$ is *initial* if, for every F-algebra (A, f), there is a unique homomorphism $h :: \mathsf{T} \to \mathsf{A}$ such that

$$h \cdot \mathsf{in}_\mathsf{T} = f \cdot \mathsf{F}\ h$$

We write $\mathsf{fold}_\mathsf{T}\ f$ for this h, giving the universal property

$$h = \mathsf{fold}_\mathsf{T}\ f \Leftrightarrow h \cdot \mathsf{in}_\mathsf{T} = f \cdot \mathsf{F}\ h$$

Thus, fold_T has type $(\mathsf{F}\ \mathsf{A} \to \mathsf{A}) \to (\mathsf{T} \to \mathsf{A})$.

The datatype definition

$$\mathsf{T} = \textsc{data}\ \mathsf{F}$$

introduces the initial F-algebra $(\mathsf{T}, \mathsf{in}_\mathsf{T})$ and the fold operator fold_T. This was generalized to polymorphic datatypes, using bifunctors rather than functors: the datatype definition

$$\mathsf{T}\ \mathsf{A} = \textsc{data}\ (\mathsf{A}\oplus)$$

introduces the initial $(\mathsf{A}\oplus)$-algebra $(\mathsf{T}\mathsf{A}, \mathsf{in}_{\mathsf{T}\ \mathsf{A}})$ and the fold operator $\mathsf{fold}_{\mathsf{T}\ \mathsf{A}}$, from which we usually omit the type subscript A, if not the whole of the subscript.

2.4.1 Example: lists

We define the datatype of lists by

$$\mathsf{List}\ \mathsf{A} = \textsc{data}\ (\underline{1} \mathbin{\hat{+}} (\underline{\mathsf{A}} \mathbin{\hat{\times}} \mathsf{Id}))$$

where $\times$ is product, $+$ is coproduct, 1 is the unit type, Id is the identity functor, underlining denotes constant functors and a superscript hat denotes lifting of a bifunctor. This induces a constructor

$$\mathsf{in}_{\mathsf{List}} :: 1 + \mathsf{A} \times \mathsf{List}\ \mathsf{A} \to \mathsf{A}$$

and a fold operator

$$\mathsf{fold}_{\mathsf{List}} :: (1 + \mathsf{B} \times \mathsf{List}\ \mathsf{A} \to \mathsf{B}) \to (\mathsf{List}\ \mathsf{A} \to \mathsf{B})$$

We introduce the syntactic sugar

$$\begin{aligned} \mathsf{const}\ [\,] \triangledown \mathit{cons} &= \mathsf{in}_{\mathsf{List}} \\ \mathit{foldr}\ f\ e \qquad &= \mathsf{fold}_{\mathsf{List}}\ (\mathsf{const}\ e \triangledown f) \end{aligned}$$

where

$$\mathsf{const}\ a\ b = a$$

This corresponds to the curried Haskell equivalents

```
[]   ::  [a]
(:)  ::  a -> [a] -> [a]
```

for the constructors, and

```
foldr  ::  (a -> b -> b) -> b -> ([a] -> b)
foldr f e []     = e
foldr f e (x:xs) = f x (foldr f e xs)
```

for the fold.

2.4.2 Example: non-empty lists

In our case study at the end of this chapter, we will also require a datatype of non-empty lists

$$\mathsf{PList}\ \mathsf{A} = \textsc{data}\ (\underline{\mathsf{A}} \mathbin{\hat{+}} (\underline{\mathsf{A}} \mathbin{\hat{\times}} \mathsf{Id}))$$

We introduce the syntactic sugar

$$\begin{aligned} \mathit{wrap} \triangledown \mathit{consp} &= \mathsf{in}_{\mathsf{PList}} \\ \mathit{foldrp}\ f\ g \qquad &= \mathsf{fold}_{\mathsf{PList}}\ (g \triangledown f) \end{aligned}$$

The Haskell standard prelude has no equivalent, but encourages the reuse of ordinary lists instead, trading notational convenience for lost structure.

2.4.3 Relational folds

The theory from Chapter 5 summarized above is fine in the categories $\mathcal{S}et$ and $\mathcal{C}po$, but turns out not to be appropriate in the category $\mathcal{R}el$ of sets and relations, because too much of the structure collapses there. In particular (Exercise 2.6.4), converses make products and coproducts coincide.

However, any functor in $\mathcal{S}et$ can be extended in a canonical way to a monotonic functor (or *relator*) on $\mathcal{R}el$. In particular, the monotonic extension of cartesian product can be used in place of the true categorical product of $\mathcal{R}el$ (which coincides with disjoint sum).

As a result, folds can be generalized to relations. For monotonically extended functor F, the initial (in $\mathcal{S}et$) F-algebra $(\mathsf{T}, \mathsf{in})$ is also initial in $\mathcal{R}el$: we still have the all-important universal property

$$h = \mathsf{fold}_\mathsf{T}\ f \Leftrightarrow h \cdot \mathsf{in}_\mathsf{T} = f \cdot \mathsf{F}\ h$$

only now the fold acts on and yields relations: $\mathsf{fold}_\mathsf{T} :: (\mathsf{F}\ \mathsf{A} \rightsquigarrow \mathsf{A}) \to (\mathsf{T} \rightsquigarrow \mathsf{A})$.

Some examples of relational folds on lists that will be used later (Exercises 3.8.5 and 3.8.9) are as follows. Recall Lambek's Lemma (§2.5.5 of Chapter 5), which states that folding with the constructors is the identity function; on lists this becomes $\mathit{foldr}\ \mathit{cons}\ [\,] = \mathsf{id}$. Folding with any subrelation of the constructors gives a coreflexive; for example, the coreflexive *ordered* that holds of ascending lists is given by

$$\mathit{ordered} = \mathit{foldr}\ (\mathit{cons} \cdot \mathit{ok})\ [\,] \quad \text{where } \mathit{ok}\ (x, xs) = \mathsf{map}\ ((x \leq)?)\ xs$$

Using the power of relations in the other direction, namely non-determinism, we can obtain simple characterizations of many combinatorial operations. For example, the relation *prefix* returns an arbitrary prefix of a list:

$$\mathit{prefix} = \mathit{fold}\ (\mathit{cons} \cup \mathsf{const}\ [\,])\ [\,]$$

and the relation *subseq* returns an arbitrary subsequence:

$$\mathit{subseq} = \mathit{foldr}\ (\mathit{cons} \cup \mathsf{exr})\ [\,]$$

2.4.4 Unfolds

One might expect now to dualize the relational generalization of folds to get relational unfolds. However, this is not necessary, because converse gives us everything we need: for functor F, the datatype $\mathsf{T} = \text{DATA}\ \mathsf{F}$ is also a codatatype, and for algebra $f :: \mathsf{FA} \rightsquigarrow \mathsf{A}$, the converse $f^\circ :: \mathsf{A} \rightsquigarrow \mathsf{FA}$ is a coalgebra, and $(\mathsf{fold}(f^\circ))^\circ$ works as an unfold (Exercise 2.6.6).

2.4.5 Fusion

Perhaps the fundamental property of folds, as illustrated at length in Chapter 5, is the fusion theorem: for $\mathsf{T} = \text{DATA}\ \mathsf{F}$,

$$h \cdot \mathsf{fold}_\mathsf{T}\ f = \mathsf{fold}_\mathsf{T}\ g \Leftarrow h \cdot f = g \cdot \mathsf{F}\ h$$

There are two variants of this fusion theorem for relational folds:

$$h \cdot \mathsf{fold}_{\mathsf{T}}\ f \subseteq \mathsf{fold}_{\mathsf{T}}\ g \Leftarrow h \cdot f \subseteq g \cdot \mathsf{F}\ h$$
$$h \cdot \mathsf{fold}_{\mathsf{T}}\ f \supseteq \mathsf{fold}_{\mathsf{T}}\ g \Leftarrow h \cdot f \supseteq g \cdot \mathsf{F}\ h$$

Of course, fusion as an equality follows from these two.

2.5 Bibliographic notes

The relational algebra has a very elegant axiomatic characterization, dating back to Tarski [30], and this is how it is often presented and used [1, 2]. Our presentation follows the axiomatization as a (unitary tabular) allegory in [5], and owes more to Freyd and Ščedrov [14] than to Tarski. (In an allegorical setting, the join $\cup$ is actually not the dual of the meet $\cap$ in general; in fact, only *locally complete* allegories have joins, and only in *boolean* locally complete allegories are they dual to meets. We gloss over these details, because we are not directly concerned with the axiomatization here.)

Using either axiomatization leads to *pointfree reasoning*, which is certainly concise, but is often difficult to follow, especially when applied to specific programming problems as opposed to general theory. This is because 'plumbing combinators' must be used to pass values around. In computer programming, we have long realized that variables are important, and that their names clarify programs; the same observation applies to the relational algebra. However, variables interact awkwardly with non-determinism, and finding the right approach is still a matter of ongoing research. The approach taken here is based on [11]; other approaches include [7, 13, 31, 25, 26, 28].

2.6 Exercises

1. Using the universal property of $\cap$ (§2.1.5), show that it is associative, commutative and idempotent:

   $$\begin{aligned} f \cap (g \cap h) &= (f \cap g) \cap h \\ f \cap g &= g \cap f \\ f \cap f &= f \end{aligned}$$

   and that monotonicity of composition can be reexpressed as follows:

   $$\begin{aligned} f \cdot (g \cap h) &\subseteq (f \cdot g) \cap (f \cdot h) \\ (f \cap g) \cdot h &\subseteq (f \cdot h) \cap (g \cdot h) \end{aligned}$$

   (that is, derive these properties from the original characterization, and vice versa).
2. Prove the shunting rules in Lemma 1.
3. The Λ operation is an isomorphism; there is a one-to-one correspondence between relations $\mathsf{A} \rightsquigarrow \mathsf{B}$ and set-valued functions $\mathsf{A} \to \mathcal{S}et\ \mathsf{B}$. What is the inverse operation? That is, given a set-valued function f of type $\mathsf{A} \to \mathcal{S}et\ \mathsf{B}$, what is the corresponding relation of type $\mathsf{A} \rightsquigarrow \mathsf{B}$? (Hint: look ahead to §3.1.)

4. A straightforward interpretation in $\mathcal{R}el$ of the standard theory of datatypes from Chapter 5 is inappropriate: products and coproducts coincide, whereas we intend them to be different constructions. Technically, this is because $\mathcal{R}el$ is its own dual, whereas $\mathcal{S}et$ and $\mathcal{C}po$ are not self-dual.
   Categorically speaking, bifunctor $\otimes$ is a product if arrows $f :: \mathsf{A} \rightsquigarrow \mathsf{B}$ and $g :: \mathsf{A} \rightsquigarrow \mathsf{C}$ uniquely determine a morphism $h :: \mathsf{A} \rightsquigarrow \mathsf{B} \otimes \mathsf{C}$ with $\mathsf{exl} \cdot h = f$ and $\mathsf{exr} \cdot h = g$. Dually, bifunctor $\oplus$ is a coproduct if arrows $f :: \mathsf{A} \rightsquigarrow \mathsf{C}$ and $g :: \mathsf{B} \rightsquigarrow \mathsf{C}$ uniquely determine a morphism $h :: \mathsf{A} \oplus \mathsf{B} \rightsquigarrow \mathsf{C}$ with $h \cdot \mathsf{inl} = f$ and $h \cdot \mathsf{inr} = g$.
   (a) Show that cartesian product is not a categorical product in $\mathcal{R}el$.
   (b) Show that disjoint sum is a categorical coproduct in $\mathcal{R}el$.
   (c) Show that disjoint sum is in fact also a categorical product in $\mathcal{R}el$.
   (d) Show, using converses, that any categorical product in $\mathcal{R}el$ is necessarily a coproduct, and vice versa.
5. A *tabulation* of relation $h :: \mathsf{A} \rightsquigarrow \mathsf{B}$ is a pair of functions $f :: \mathsf{C} \rightsquigarrow \mathsf{A}$, $g :: \mathsf{C} \rightsquigarrow \mathsf{B}$ such that $h = g \cdot f^\circ$ and $(f^\circ \cdot f) \cap (g^\circ \cdot g) = \mathsf{id}$. Assuming that every relation has a tabulation, show that for relator F:
   (a) when h is a function, so is $\mathsf{F}\ h$, and $(\mathsf{F}\ h)^\circ = \mathsf{F}\ (h^\circ)$;
   (b) a functor is a relator iff $(\mathsf{F}h)^\circ = \mathsf{F}(h^\circ)$ for any (not necessarily functional) h;
   (c) if two relators F, G agree on functions, then they agree on all relations.
   Thus, functors in $\mathcal{S}et$ can be extended in a canonical way to relators in $\mathcal{R}el$.
6. Show that datatype $\mathsf{T} = \text{DATA}\ \mathsf{F}$ is also a codatatype, with final coalgebra $(\mathsf{T}, \mathsf{in}^\circ)$. That is, for algebra $f :: \mathsf{F}\,\mathsf{A} \rightsquigarrow \mathsf{A}$, show that $(\mathsf{fold}_\mathsf{T}\,(f^\circ))^\circ$ is an unfold:

   $$h = (\mathsf{fold}_\mathsf{T}\,(f^\circ))^\circ \ \Leftrightarrow\ \mathsf{in}^\circ \cdot h = \mathsf{F}\ h \cdot f$$

   This is another consequence of $\mathcal{R}el$'s self-duality.
7. Using relations, we can give very nice equational proofs of two theorems presented in Exercises 2.9.9 and 2.9.12 of Chapter 5. Despite these being properties of total functions, it seems that their simplest proofs are in terms of relations. For datatype $\mathsf{T} = \text{DATA}\ \mathsf{F}$, show that function $h :: \mathsf{T} \to \mathsf{A}$ is a fold iff $h \cdot \mathsf{in} \cdot \mathsf{F}\ h^\circ$ is simple, and function $g :: \mathsf{A} \to \mathsf{T}$ is an unfold iff $\mathsf{F}\ g^\circ \cdot \mathsf{in}^\circ \cdot g$ is entire. Explain why these conditions are equivalent to those from Chapter 5.
8. Prove the two inclusion fusion theorems

   $$h \cdot \mathsf{fold}_\mathsf{T}\ f \subseteq \mathsf{fold}_\mathsf{T}\ g \Leftarrow h \cdot f \subseteq g \cdot \mathsf{F}\ h$$
   $$h \cdot \mathsf{fold}_\mathsf{T}\ f \supseteq \mathsf{fold}_\mathsf{T}\ g \Leftarrow h \cdot f \supseteq g \cdot \mathsf{F}\ h$$

3 Optimization problems

Optimization problems are typically of the form 'select the best construct generated by this means, which satisfies that test'. Such a specification is executable, provided that only finitely many constructs are generated, but it is in general too expensive to compute because there still many constructs to consider. The algorithm can be improved by promoting the test and the selection inside the

generation, thereby avoiding having to generate and eliminate constructs that cannot possibly contribute to the optimal solution.

Sometimes the improved algorithm has exactly the same extensional behaviour as the original specification, but is faster to execute. More often, however, the specification is non-deterministic (there may be several optimal solutions), and the improved algorithm is a *refinement* of the specification, yielding only a subset of the optimal solutions. The missing solutions are pruned to permit a more efficient algorithm; correctness is maintained by ensuring that for every solution pruned, an equally good one is retained. This refinement characteristic is a strong motivation for the move from functional to relational programming: refinement makes little sense with total functions.

In this chapter we will consider a restricted, but still large, class of optimization problems, of the form

$$\mathsf{min}\ r \cdot \Lambda(\mathsf{fold}\ f)$$

The 'test' phase is omitted, and the feasible constructs are generated directly by a (relational) fold. We take the breadth of this fold to get the set of all feasible constructs, and select from this set a minimal element under the preorder r.

We will develop two kinds of improved algorithm for such problems, depending on characteristics of the ingredients f and r. The first embodies a *greedy* approach, building a single optimal solution step by step. The second embodies a *thinning* approach, building up a (hopefully small) collection of representative solutions, some unforeseeable one of which will lead to the optimal solution. The second approach assumes weaker conditions on the ingredients, but yields a less efficient algorithm.

3.1 The Eilenberg-Wright Lemma

Let $choose :: \mathsf{Set}\ \mathsf{A} \rightsquigarrow \mathsf{A}$ denote the membership relation for sets, so that

$$x \mathrel{\reflectbox{$\rightsquigarrow$}} choose\ xs \Leftrightarrow x \in xs$$

Then we have the following lemma:

Lemma 2 (Eilenberg-Wright). *For* $\mathsf{T} = \text{DATA}\ \mathsf{F}$*,*

$$\Lambda(\mathsf{fold}_{\mathsf{T}}\ f) = \mathsf{fold}_{\mathsf{T}}\ (\Lambda(f \cdot \mathsf{F}\ choose))$$

Informally, the set of results returned by a relational fold can be obtained as a functional fold that at each stage returns the set of all possible partial results.

In particular, we have that

$$\mathsf{min}\ r \cdot \Lambda(\mathsf{fold}_{\mathsf{T}}\ f) = \mathsf{min}\ r \cdot \mathsf{fold}_{\mathsf{T}}\ (\Lambda(f \cdot \mathsf{F}\ choose))$$

That is, rather than computing a single complete construct in all possible ways and taking the best of the results, we can explore all choices at each step, building up all constructs 'in parallel'.

3.2 Preorders

A preorder $r :: \mathsf{A} \leadsto \mathsf{A}$ is a relation that is

- *reflexive*, that is, $\mathsf{id} \subseteq r$, and
- *transitive*, that is, $r \cdot r \subseteq r$.

In addition, preorder $r :: \mathsf{A} \leadsto \mathsf{A}$ may or may not be

- *connected*, that is, $r \cup r^\circ = \mathsf{A} \times \mathsf{A}$.

When we come to translate our results into Haskell, it will be convenient to represent a preorder r by its *characteristic function* $\chi(r)$ of type $\mathsf{A} \times \mathsf{A} \to \mathsf{Bool}$ (or perhaps its curried version $\mathsf{A} \to \mathsf{A} \to \mathsf{Bool}$) such that $x \mathrel{\reflectbox{$\leadsto$}} r\ y \Leftrightarrow \chi(r)\,(x, y)$. With this representation, the properties become more long-winded but perhaps more familiar:

- a preorder r is *reflexive*, that is, $\chi(r)\,(x, x)$ for every $x \in \mathsf{A}$;
- a preorder r is *transitive*, that is, $\chi(r)\,(x, y) \wedge \chi(r)\,(y, z) \Rightarrow \chi(r)\,(x, z)$ for every $x, y, z \in \mathsf{A}$;
- a preorder r may or may not be *connected*, that is, $\chi(r)\,(x, y) \vee \chi(r)\,(y, x)$ for every $x, y \in \mathsf{A}$.

3.3 Monotonicity

We say that that an F-algebra $f :: \mathsf{F}\,\mathsf{A} \leadsto \mathsf{A}$ is *monotonic under a preorder* r if

$$f \cdot \mathsf{F}\,r \subseteq r \cdot f$$

Equivalently, in terms of characteristic functions, we have that f is monotonic under r if $\chi(\mathsf{F}\,r)\,(x, y)$ and $u \mathrel{\reflectbox{$\leadsto$}} f\,x$ together imply that there exists a $v \mathrel{\reflectbox{$\leadsto$}} f\,y$ such that $\chi(r)\,(u, v)$. (Note that $\mathsf{F}\,r$ is a relation of type $\mathsf{F}\,\mathsf{A} \leadsto \mathsf{F}\,\mathsf{A}$, so its characteristic function $\chi(\mathsf{F}\,r)$ has type $\mathsf{F}\,\mathsf{A} \times \mathsf{F}\,\mathsf{A} \to \mathsf{Bool}$.)

For example, addition of naturals $plus :: \mathsf{Pair\ Nat} \to \mathsf{Nat}$ is monotonic under leq, the normal linear ordering, because $plus \cdot \mathsf{Pair}\ leq \subseteq leq \cdot plus$, or in terms of points, $a \mathrel{\reflectbox{$\leadsto$}} leq\ c$ and $b \mathrel{\reflectbox{$\leadsto$}} leq\ d$ imply that $plus\ (a, b) \mathrel{\reflectbox{$\leadsto$}} leq\ (plus\ (c, d))$.

3.4 Minimum

The function $\mathsf{min} :: (\mathsf{A} \leadsto \mathsf{A}) \to (\mathsf{Set}\,\mathsf{A} \leadsto \mathsf{A})$ is defined by

$$\mathsf{min}\ r = choose \cap (r \mathbin{/} choose^\circ)$$

Informally, $\mathsf{min}\ r$ takes a set xs and returns a value x that both is an element of xs and satisfies $x \mathrel{\reflectbox{$\leadsto$}} r\ y$ for every $y \mathrel{\reflectbox{$\leadsto$}} choose\ xs$; in points,

$$x \mathrel{\reflectbox{$\leadsto$}} \mathsf{min}\ r\ xs \ \Leftrightarrow\ (x \in xs) \wedge (\forall y \in xs.\ \chi(r)\,(x, y))$$

Of course, this definition is perfectly symmetric: to obtain the maximum under a preorder r, simply compute the minimum under r°.

If every non-empty set has a minimum under a preorder r, then r is necessarily connected. In general, therefore, we will only be interested in uses of $\mathsf{min}\, r$ for connected preorders r.

When it comes to implementing this operator in Haskell, we first must decide how to represent sets. We choose the simplest representation, as an unordered list, but because we will be computing minimum elements of sets, we will stick to non-empty lists; so we define the function *minlist* as follows:

$$\begin{aligned} &\mathit{minlist} :: (\mathsf{A} \times \mathsf{A} \to \mathsf{Bool}) \to (\mathsf{PList}\,\mathsf{A} \to \mathsf{A}) \\ &\mathit{minlist}\ r = \mathit{foldrp}\ m\ \mathsf{id} \quad \text{where } m\ (x, y) = \text{if } r\ (x, y) \text{ then } x \text{ else } y \end{aligned}$$

This implementation breaks ties in favour of the leftmost of two equivalent elements. If r is connected, then

$$\mathit{minlist}\ (\chi(r)) \subseteq \mathsf{min}\ r \cdot \mathsf{setify}$$

where $\mathsf{setify} :: \mathsf{PList}\,\mathsf{A} \to \mathsf{Set}\,\mathsf{A}$ converts a non-empty list to the set of its elements.

3.5 The Greedy Theorem

Greedy algorithms for our class of optimization problems are captured by the following theorem.

Theorem 3 (Greedy Theorem). *Suppose* $\mathsf{T} = \text{DATA}\ \mathsf{F}$, *and* F*-algebra* $f :: \mathsf{F}\,\mathsf{A} \rightsquigarrow \mathsf{A}$ *is monotonic under the preorder* r°. *Then*

$$\mathsf{fold}_\mathsf{T}\ (\mathsf{min}\ r \cdot \Lambda f) \subseteq \mathsf{min}\ r \cdot \Lambda(\mathsf{fold}_\mathsf{T}\ f)$$

In fact, we have a stronger theorem, of which the above is a simple corollary.

Theorem 4 (Refining Greedy Theorem). *Suppose* $\mathsf{T} = \text{DATA}\ \mathsf{F}$, *and* F*-algebra* $f :: \mathsf{F}\,\mathsf{A} \rightsquigarrow \mathsf{A}$ *is monotonic under the preorder* q°, *where* $q \subseteq r$. *Then*

$$\mathsf{fold}_\mathsf{T}\ (\mathsf{min}\ q \cdot \Lambda f) \subseteq \mathsf{min}\ r \cdot \Lambda(\mathsf{fold}_\mathsf{T}\ f)$$

Proof (Sketch). Using the Eilenberg-Wright Lemma, it suffices to show

$$\mathsf{fold}_\mathsf{T}\ (\mathsf{min}\ q \cdot \Lambda f) \subseteq \mathsf{min}\ r \cdot \mathsf{fold}_\mathsf{T}\ (\Lambda(f \cdot \mathsf{F}\ \mathit{choose}))$$

This in turn follows, by fusion, from

$$\mathsf{min}\ q \cdot \Lambda f \cdot \mathsf{F}\ (\mathsf{min}\ r) \subseteq \mathsf{min}\ r \cdot \Lambda(f \cdot \mathsf{F}\ \mathit{choose})$$

Discharging this final proof obligation is left as an exercise for the reader (Exercise 3.8.4).

Informally, to say that f is monotonic under r° is to say that for any 'input' $x :: \mathsf{F}\,\mathsf{A}$, any 'worse input' $y \leftsquigarrow \mathsf{F}\ r^\circ\ x$, and any result $b \leftsquigarrow f\ y$ of f on this worse input, there corresponds a result $a \leftsquigarrow f\ x$ of f on the original input that is 'better' than b, that is, $b \leftsquigarrow r^\circ\ a$. Thus, degrading the inputs to f will always degrade the output, or conversely, it is never beneficial to pick suboptimal intermediate results, as they will only lead to suboptimal final results. Overall, the Greedy Theorem states that a minimal result can be computed by maintaining a single

minimal partial result at each stage of the folding process, and so it lies at the other extreme to the Eilenberg-Wright Lemma, which embodies a kind of exhaustive search.

The Refining Greedy Theorem is a little more flexible. It is sometimes the case that f fails to be monotonic under r°, but enjoys monotonicity under a refined ordering q° where $q \subseteq r$. We will see an example in §4.5. Provided that q remains connected, the relation $\mathsf{min}\ q$ is entire (except for empty sets). The greedy algorithm itself will also be entire (except for empty sets), refining the plain greedy algorithm from Theorem 3: it computes an optimal solution under q, and any optimal solution under q will also be optimal under r — but the converse does not hold. If q is not connected, the theorem still holds, but the resulting algorithm will not be entire (that is, will not always yield a result).

In order to implement the method as a functional program, we have to find some function $step :: \mathsf{F}\,\mathsf{A} \to \mathsf{A}$ such that $step \subseteq \mathsf{min}\,q \cdot \Lambda f$. We will give an example application of the Greedy Theorem in §4.

3.6 Thinning

Monotonicity under a connected preorder is a strong condition that is satisfied in very few optimization problems; and indeed, few such problems admit greedy solutions. More useful would be something between the two extremes of the Eilenberg-Wright Lemma and the Greedy Theorems, involving maintaining some but not all partial solutions.

For a not necessarily connected preorder q on A, the relation $\mathsf{thin}\ q :: \mathsf{Set}\,\mathsf{A} \rightsquigarrow \mathsf{Set}\,\mathsf{A}$ takes a set xs and returns some subset ys of xs with the property that all elements of xs have a lower bound under q in ys. More precisely,

$$\mathsf{thin}\ q = (choose \setminus choose)\ \cap\ ((choose^\circ \cdot q)\ /\ choose^\circ)$$

If $ys \leftsquigarrow \mathsf{thin}\ q\ xs$, then the first conjunct here says that ys is a subset of xs, and the second says that for every $x \leftsquigarrow choose\ xs$, there is a $y \leftsquigarrow choose\ ys$ with $y \leftsquigarrow q\ x$; in points,

$$ys \leftsquigarrow \mathsf{thin}\ q\ xs\ \Leftrightarrow\ (ys \subseteq xs) \wedge (\forall x \in xs.\ \exists y \in ys.\ \chi(q)\,(x, y))$$

The following lemma allows us to introduce applications of thin.

Lemma 5 (Thin introduction). *Provided that q and r are both preorders and $q \subseteq r$, we have*

$$\mathsf{min}\ r = \mathsf{min}\ r \cdot \mathsf{thin}\ q$$

That is, any required minimum can be obtained by selecting that minimum from a suitably thinned set. For the proof, see Exercise 3.8.6.

This leads us towards the following *Thinning Theorem*, which entails maintaining a representative collection of partial solutions at each stage of the folding process.

Theorem 6 (Thinning). *Suppose* $\mathsf{T} = \text{DATA}\ \mathsf{F}$, *and* F*-algebra* $f :: \mathsf{F}\,\mathsf{A} \rightsquigarrow \mathsf{A}$ *is monotonic under a (not necessarily connected) preorder q°, where $q \subseteq r$. Then*

$$\mathsf{min}\ r \cdot \mathsf{fold_T}\ (\mathsf{thin}\ q \cdot \Lambda(f \cdot \mathsf{F}\ \mathit{choose})) \subseteq \mathsf{min}\ r \cdot \Lambda(\mathsf{fold_T}\ f)$$

Proof (Sketch). We have $\mathsf{min}\ r = \mathsf{min}\ r \cdot \mathsf{thin}\ q$, so it suffices to show that

$$\mathsf{fold_T}\ (\mathsf{thin}\ q \cdot \Lambda(f \cdot \mathsf{F}\ \mathit{choose})) \subseteq \mathsf{thin}\ q \cdot \Lambda(\mathsf{fold_T}\ f)$$

Just like the Greedy Theorem, this latter inclusion can be proved (Exercise 3.8.7) by making use of the Eilenberg-Wright Lemma and fusion.

3.6.1 Implementation

In order to implement the thinning method as a functional program, we have to implement

$$\mathsf{thin}\ q \cdot \Lambda(f \cdot \mathsf{F}\ \mathit{choose}) :: \mathsf{F}\ (\mathsf{Set\ A}) \rightsquigarrow \mathsf{Set\ A}$$

It seems reasonable to represent these sets as lists, sorted by some connected preorder (but not necessarily by q itself); we can capitalize on the list ordering to implement the thinning step efficiently. Therefore, we will actually construct a function

$$\mathit{step} :: \mathsf{F}\ (\mathsf{PList\ A}) \to \mathsf{PList\ A}$$

such that

$$\mathsf{setify} \cdot \mathit{step} \subseteq \mathsf{thin}\ q \cdot \Lambda(f \cdot \mathsf{F}\ \mathit{choose}) \cdot \mathsf{F\ setify}$$

To do this, we will need a number of ingredients:

- a function

  $$\mathit{sort} :: (\mathsf{A} \times \mathsf{A} \to \mathsf{Bool}) \to (\mathsf{Set\ A} \to \mathsf{PList\ A})$$

  that sorts a finite set under a given connected preorder;
- a function

  $$\mathit{mergelists} :: (\mathsf{A} \times \mathsf{A} \to \mathsf{Bool}) \to (\mathsf{PList\ (PList\ A)} \to \mathsf{PList\ A})$$

  that merges a list of sorted lists into a single sorted list;
- a function

  $$\mathit{thinlist} :: (\mathsf{A} \times \mathsf{A} \to \mathsf{Bool}) \to (\mathsf{PList\ A} \to \mathsf{PList\ A})$$

  that implements thin:

  $$\mathsf{setify} \cdot \mathit{thinlist}\ (\chi(q)) \subseteq \mathsf{thin}\ q \cdot \mathsf{setify}$$

  on a suitably sorted list in a reasonably efficient manner, that is, quickly (in linear time) and effectively (yielding a short result), and does so stablely:

  $$\mathit{thinlist}\ (\chi(q)) \subseteq \mathit{subseq}$$
- a function

  $$\mathit{cp_F} :: \mathsf{F}\ (\mathsf{PList\ A}) \to \mathsf{PList\ (F\ A)}$$

  that converts an F-structure of lists (possibly of differing lengths) into a list of F-structures (all of the same shape);

- a function $combine :: \mathsf{F}\ \mathsf{A} \to \mathsf{PList}\ \mathsf{A}$ satisfying

  $$combine \subseteq sort\ (\chi(p)) \cdot \mathsf{thin}\ q \cdot \Lambda f$$

  for some connected preorder p.

Given these ingredients, we define

$$step = thinlist\ (\chi(q)) \cdot mergelists\ (\chi(p)) \cdot \mathsf{map}\ combine \cdot cp_{\mathsf{F}}$$

and claim (Exercise 3.8.8) that

$$minlist\ (\chi(r)) \cdot \mathsf{fold}\ step \subseteq \mathsf{min}\ r \cdot \mathsf{fold}\ (\mathsf{thin}\ q \cdot \Lambda(f \cdot \mathsf{F}\ choose))$$

There is no constraint on p for correctness, but for efficiency a suitable choice will bring related elements together to allow effective thinning with $thinlist\ (\chi(q))$.

3.7 Bibliographic notes

The Eilenberg-Wright Lemma, and indeed relational folds themselves, were first employed to reason about the equivalence of deterministic and non-deterministic automata [12].

Greedy algorithms are well-known in the algorithm design community; an in-depth study of their properties is in [21], and a more pragmatic description with numerous examples in [8]. Thinning algorithms as an abstraction in their own right are due to Bird and de Moor [4, 5]. More results on greedy, thinning and dynamic programming algorithms for optimization problems are given in Curtis' thesis [9]; there the emphasis is on algorithms in which the generation phase is less structured, a simple loop rather than a fold or an unfold.

Cross products (§3.6.1) and the related zips are investigated in great depth in Hoogendijk's thesis [19].

3.8 Exercises

1. Prove the Eilenberg-Wright Lemma (§3.1), using the universal property of folds.
2. Find an F-algebra $f :: \mathsf{F}\ \mathsf{A} \rightsquigarrow \mathsf{A}$ and a preorder r on A such that f is monotonic under r but not under r°. Prove that when f is a total function, f is monotonic under r iff it is monotonic under r°.
3. Why is connectedness of r necessary for the function *minlist* from §3.4 to be an implementation of min?
4. Complete the proof of the Refining Greedy Theorem (Theorem 4) from §3.5, by showing that

   $$\mathsf{min}\ q \cdot \Lambda f \cdot \mathsf{F}\ (\mathsf{min}\ r) \subseteq \mathsf{min}\ r \cdot \Lambda(f \cdot \mathsf{F}\ choose)$$

   where f is monotonic under q° with $q \subseteq r$.
5. The function $takewhile :: (\mathsf{A} \rightsquigarrow \mathsf{A}) \to (\mathsf{List}\ \mathsf{A} \rightsquigarrow \mathsf{List}\ \mathsf{A})$ yields the longest prefix of its second argument, all of whose elements 'satisfy' the coreflexive first argument:

   $$takewhile\ p = longest \cdot \Lambda(\mathsf{map}\ p \cdot prefix)$$

where *prefix* is defined in §2.4.3, and

$$longest = \mathsf{min}\ (length^{\circ} \cdot geq \cdot length)$$

and where *length* returns the length of a list, and *geq* is the usual linear ordering on naturals. For example, *takewhile prime* $[2,3,4,5] = [2,3]$. Use fusion to write $\mathsf{map}\ p \cdot prefix$ as a fold, and hence use the Greedy Theorem to derive the standard implementation of *takewhile* p as a fold.

6. Prove that

$$\mathsf{min}\ r \supseteq \mathsf{min}\ r \cdot \mathsf{thin}\ q$$

provided that q, r are preorders, and $q \subseteq r$. Because $\mathsf{thin}\ q \supseteq \mathsf{id}$ too, this proves Lemma 5.

7. Complete the proof of the Thinning Theorem from §3.6, by showing that

$$\mathsf{fold_T}\ (\mathsf{thin}\ q \cdot \Lambda(f \cdot \mathsf{F}\ choose)) \subseteq \mathsf{thin}\ q \cdot \Lambda(\mathsf{fold_T}\ f)$$

where $\mathsf{T} = \text{DATA}\ \mathsf{F}$ and f is monotonic under q°.

8. Justify the claim (§3.6.1) that the thinning algorithm

$$minlist\ (\chi(r)) \cdot \mathsf{fold}\ (thinlist\ (\chi(q)) \cdot mergelists\ (\chi(p)) \cdot \mathsf{map}\ combine \cdot cp_{\mathsf{F}})$$

is indeed a refinement of the optimization problem.

9. The *longest upsequence* problem [15] is to compute the longest ascending subsequence of a list; for example, the longest upsequence of $[1,6,5,2,4,3,3]$ is $[1,2,3,3]$. Formally, the problem is to compute

$$longest \cdot \Lambda(ordered \cdot subseq)$$

where *longest* is as in Exercise 3.8.5, and *ordered* and *subseq* as in §2.4.3. Derive a thinning algorithm to solve this problem.

10. In a sense, the Thinning Theorem always applies: one can always choose $q = \mathsf{id}$. Prove that this choice satisfies the conditions of Theorem 6, and that the algorithm obtained by applying the theorem is the same as the problem specification $\mathsf{min}\ r \cdot \Lambda(\mathsf{fold}\ f)$.

11. At the other extreme, the Greedy Theorem is an instance of the Thinning Theorem: choosing $q = r$ in the Thinning Theorem gives 'essentially' the greedy algorithm. More precisely, the thinning algorithm so obtained will still return a set of elements, but all will be optimal; taking the breadth gives the corresponding greedy algorithm. Justify this claim. (Thus, 'thinning' encompasses the whole spectrum from maintaining all partial solutions to maintaining just one.)

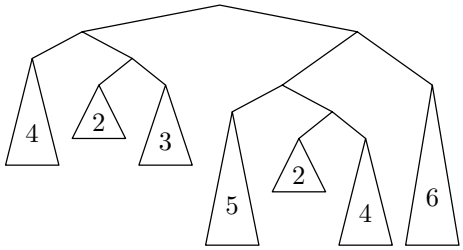
4 Optimal bracketing

To illustrate the foregoing theory, consider the $(\mathsf{A}, \leq, \oplus)$ *bracketing problem*, defined as follows. Given a linearly ordered set $(\mathsf{A}, \leq)$, an operator $\oplus :: \mathsf{A} \times \mathsf{A} \to \mathsf{A}$, and a non-empty list $a_1, a_2, \ldots a_n$ of values of type A, it is required to bracket the expression

$$a_1 \oplus a_2 \oplus \cdots \oplus a_n$$

in such a way that the result is as small as possible under the ordering $\leq$. Of course, if $\oplus$ is associative then all bracketings give equally small results.

Example 7. With $a \oplus b = max\ (a, b) + 1$, the $(\mathsf{Nat}, \leq, \oplus)$ bracketing problem corresponds to the task of combining a list of trees (with given heights $a_1, \ldots a_n$) into a single tree of minimum height. For example, given subtrees with heights $[4, 2, 3, 5, 2, 4, 6]$, one optimal tree is



Example 8. Similarly, one interpretation of the $(\mathsf{Nat} \times \mathsf{Nat}, \leq, \oplus)$ bracketing problem, where $\leq$ is the lexical ordering on pairs and

$$(c_1, l_1) \oplus (c_2, l_2) = (c_1 + c_2 + l_1 \times l_2, l_1 + l_2)$$

corresponds to the task of multiplying a list of decimals in the shortest possible time (without exploiting the commutativity of multiplication), where multiplying decimals of lengths l_1 and l_2 costs $l_1 \times l_2$, and yields a decimal of length $l_1 + l_2$.

Example 9. The $(\mathsf{Nat} \times (\mathsf{Nat} \times \mathsf{Nat}), \leq, \oplus)$ bracketing problem, where $\leq$ is the lexical ordering and

$$(c_1, (p, q)) \oplus (c_2, (q, r)) = (c_1 + c_2 + p \times q \times r, (p, r))$$

corresponds to the problem of multiplying a sequence of conformant matrices with minimal cost, where multiplying a $p \times q$ matrix by a $q \times r$ matrix costs $p \times q \times r$, and yields a $p \times r$ matrix.

4.1 Representation

We will represent a single bracketing as a binary tree of type

$$\mathsf{Tree\ A} = \textsc{data}\ (\underline{\mathsf{A}} \mathbin{\hat{+}} (\mathsf{Id} \mathbin{\hat{\times}} \mathsf{Id}))$$

for which we introduce the syntactic sugar

$$\begin{aligned} \mathit{tip}\ a &= \mathsf{in}\ (\mathsf{inl}\ a) \\ \mathit{fork}\ (t, u) &= \mathsf{in}\ (\mathsf{inr}\ (t, u)) \end{aligned}$$

for the constructors, and

$$\begin{aligned} \mathit{foldT} &:: (\mathsf{B} \times \mathsf{B} \rightsquigarrow \mathsf{B}) \to (\mathsf{A} \rightsquigarrow \mathsf{B}) \to (\mathsf{Tree\ A} \rightsquigarrow \mathsf{B}) \\ \mathit{foldT}\ f\ g &= \mathsf{fold}_{\mathsf{Tree}}\ (g \triangledown f) \end{aligned}$$

for the fold.

In particular, the function $\mathit{flatten} :: \mathsf{Tree\ A} \to \mathsf{PList\ A}$ is defined by

$$\mathit{flatten} = \mathit{foldT}\ (+\!\!+)\ \mathit{wrap}$$

The $(\mathsf{A}, \leq, \oplus)$ bracketing problem can now be formalized as one of computing

$$\mathsf{min}\ r \cdot \Lambda(\mathit{flatten}^\circ)$$

where $\mathit{cost} = \mathit{foldT}\ (\oplus)\ \mathsf{id}$, and $r = \mathit{cost}^\circ \cdot \mathit{leq} \cdot \mathit{cost}$, or in points, $\chi(r)\ (x, y) = \mathit{cost}\ x \leq \mathit{cost}\ y$ (so r is connected). Note that $\mathit{flatten}^\circ$ is partial, giving a result only on non-empty lists.

4.2 The Converse-of-a-Function Theorem

One could start by formulating methods for computing

$$\mathsf{min}\ r \cdot \Lambda((\mathsf{fold}\ f)^\circ)$$

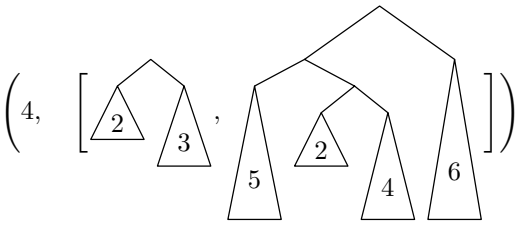
We will not take this approach (but see [5]); instead, we will make use of the following theorem to express $\mathit{flatten}^\circ$ as a fold:

Theorem 10 (Converse of a function). *Suppose* $f :: \mathsf{A} \to \mathsf{T}$ *(in particular,* f *is a total function), where* $\mathsf{T} = \text{DATA}\ \mathsf{F}$. *Furthermore, suppose* $g :: \mathsf{F}\ \mathsf{A} \rightsquigarrow \mathsf{A}$ *is surjective and satisfies* $f \cdot g \subseteq \mathsf{in} \cdot \mathsf{F}\ f$. *Then* $f^\circ = \mathsf{fold}_\mathsf{T}\ g$.

Use of this theorem is a definite design step in solving an optimization problem: it prescribes the structure of the resulting program, and thereby rules out some potential solutions. It is therefore not always the right step to take; however, it turns out to be a productive one for some instances of the bracketing problem.

4.3 Spines

To express $\mathit{flatten}^\circ$ as a fold, we will represent trees by their *left spines*. The left spine of a tree is a pair consisting of the leftmost tip value and the sequence of right subtrees along the path from the leftmost tip to the root. Thus we define $\mathsf{Spine}\ \mathsf{A} = \mathsf{A} \times \mathsf{List}\ (\mathsf{Tree}\ \mathsf{A})$. For example, the optimal tree in Example 7 has the left spine



The bijection *roll* takes a spine $(a, [t_1, t_2, \ldots, t_n])$ and returns the tree

$$\mathit{fork}\ (\cdots \mathit{fork}\ (\mathit{fork}\ (\mathit{tip}\ a, t_1), t_2) \cdots, t_n)$$

Formally, we have

$$\mathit{roll}\ (a, ts) = \mathit{foldl}\ \mathit{fork}\ (\mathit{tip}\ a)\ ts$$

where *foldl* is the natural fold function for *snoc lists*, that is, lists built by adding elements to the end rather than the beginning. On ordinary lists we can simulate *foldl* by

$$\begin{aligned} &\mathit{foldl}\ f\ e\ [\,] &&= e \\ &\mathit{foldl}\ f\ e\ (\mathit{cons}\ (x, xs)) &&= \mathit{foldl}\ f\ (f\ (e, x))\ xs \end{aligned}$$

Using the Converse-of-a-Function Theorem, we can now obtain (Exercise 4.10.1)

$$(\mathit{flatten} \cdot \mathit{roll})^\circ = \mathit{foldrp}\ \mathit{add}\ \mathit{one}$$

where $\mathit{one}\ a = (a, [\,])$ and $\mathit{add} :: \mathsf{A} \times \mathsf{Spine\ A} \rightsquigarrow \mathsf{Spine\ A}$ is defined (using a non-injective pattern) by

$$\mathit{add}\ (a, (b, xs +\!\!+ ys)) \mathrel{\widehat{=}} (a, \mathit{cons}\ (\mathit{roll}\ (b, xs), ys))$$

For example,

$$\begin{aligned} &\mathit{foldrp}\ \mathit{add}\ \mathit{one}\ [1, 2, 3] \\ =\ &\mathit{add}\ (1, \mathit{add}\ (2, \mathit{one}\ 3)) \\ =\ &\mathit{add}\ (1, \mathit{add}\ (2, (3, [\,]))) \\ =\ &\mathit{add}\ (1, (2, [\mathit{tip}\ 3])) \\ =\ &(1, [\mathit{tip}\ 2, \mathit{tip}\ 3]) \,\square\, (1, [\mathit{fork}\ (\mathit{tip}\ 2, \mathit{tip}\ 3)]) \end{aligned}$$

That is, both

$$(1, [\mathit{tip}\ 2, \mathit{tip}\ 3]) \mathrel{\leftsquigarrow} \mathit{foldrp}\ \mathit{add}\ \mathit{one}\ [1, 2, 3]$$

and

$$(1, [\mathit{fork}\ (\mathit{tip}\ 2, \mathit{tip}\ 3)]) \mathrel{\leftsquigarrow} \mathit{foldrp}\ \mathit{add}\ \mathit{one}\ [1, 2, 3]$$

and nothing else is a possible result.

Now we reason, for any r,

$$\begin{aligned} &\mathsf{min}\ r \cdot \Lambda(\mathit{flatten}^\circ) \\ =\ &\quad \{\ \mathit{roll} \text{ is a surjective function, so } \mathit{roll} \cdot \mathit{roll}^\circ = \mathsf{id}\ \} \\ &\mathsf{min}\ r \cdot \Lambda(\mathit{roll} \cdot \mathit{roll}^\circ \cdot \mathit{flatten}^\circ) \\ =\ &\quad \{\ \text{claim, with } \chi(rr)\ (x, y) = \chi(r)\ (\mathit{roll}\ x, \mathit{roll}\ y)\ \} \\ &\mathit{roll} \cdot \mathsf{min}\ rr \cdot \Lambda(\mathit{roll}^\circ \cdot \mathit{flatten}^\circ) \\ =\ &\quad \{\ \text{converse of a function}\ \} \\ &\mathit{roll} \cdot \mathsf{min}\ rr \cdot \Lambda(\mathit{foldrp}\ \mathit{add}\ \mathit{one}) \end{aligned}$$

The claim above is that computing a minimal tree under r is equivalent to computing a minimal spine under rr, because *roll* is a bijection between the two types.

So our problem now is to compute $\mathsf{min}\ rr \cdot \Lambda(\mathit{foldrp}\ \mathit{add}\ \mathit{one})$, where preorder rr on $\mathsf{Spine\ A}$ is defined by

$$\chi(rr)\ (x, y) = \mathit{cost}\ (\mathit{roll}\ x) \leq \mathit{cost}\ (\mathit{roll}\ y)$$

4.4 An application of thinning

For most instances of the bracketing problem, the algebra $\mathit{one} \triangledown \mathit{add}$ is not monotonic under rr°. For Example 7, for instance, the two trees $\mathit{fork}(\mathit{tip}1, \mathit{fork}(\mathit{tip}1, \mathit{tip}1))$ and $\mathit{fork}\,(\mathit{fork}\,(\mathit{tip}\,1, \mathit{tip}\,1), \mathit{tip}\,1)$ that can be built from the list of heights $[1, 1, 1]$ are equally good under r (and so their spines are equally good under rr), but

only the former can be extended by *add* to make the unique optimal tree for *cons* $(1, [1, 1, 1])$. So the Greedy Theorem does not apply to Example 7.

However, some instances of the bracketing problem succumb to a thinning algorithm based on the 'pairwise ordering' of the left spines of the trees, defined as follows. Let

$$\begin{aligned} &\mathit{lspinecosts}\ (a, ts) = \mathsf{map}\ (\mathit{cost}'\ a)\ (\mathit{prefixes}\ ts) \\ &\mathit{cost}'\ a\ us \qquad\quad = \mathit{cost}\ (\mathit{roll}\ (a, us)) \end{aligned}$$

where *prefixes* returns the prefixes of a list in order of decreasing length — for example,

$$\mathit{prefixes}\ [1, 2, 3] = [[1, 2, 3], [1, 2], [1], [\,]]$$

Then we choose the preorder q such that

$$\chi(q)\ (x, y) = \mathit{lspinecosts}\ x \preccurlyeq \mathit{lspinecosts}\ y$$

where $\preccurlyeq$ is *pairwise* ordering of sequences: $[a_1, \ldots, a_m] \preccurlyeq [b_1, \ldots, b_n]$ if and only if $m \leq n$ and $a_i \leq b_i$ for $1 \leq i \leq m$. Informally, $as \preccurlyeq bs$ when as is no longer than bs, and each element of as is at most the corresponding element of bs. Note that this q is not connected. Note also that $q \subseteq rr$: because $\mathit{cost}\ (\mathit{roll}\ x) = \mathit{head}\ (\mathit{lspinecosts}\ x)$, if $x \leftsquigarrow q\ y$ then certainly $\mathit{cost}\ (\mathit{roll}\ x) \leq \mathit{cost}\ (\mathit{roll}\ y)$.

Now we claim that, for Examples 7 and 9, the algebra $\mathit{one} \triangledown \mathit{add}$ is monotonic under this q°. We will leave the proof of monotonicity as Exercise 4.10.5 for the energetic reader. We will merely observe here that for the minimum height tree problem, the spines of the two trees $\mathit{fork}\ (\mathit{tip}\ 1, \mathit{fork}\ (\mathit{tip}\ 1, \mathit{tip}\ 1))$ and $\mathit{fork}\ (\mathit{fork}\ (\mathit{tip}\ 1, \mathit{tip}\ 1), \mathit{tip}\ 1)$ introduced above are definitely not equally good under q: the costs of the reverse prefixes of the two spines are $[3, 1]$ and $[3, 2, 1]$, and the former is pairwise strictly less than the latter.

Therefore we conclude that, at least for these two examples,

$$\begin{aligned} & \mathsf{min}\ rr \cdot \Lambda(\mathit{flatten}^\circ) \\ = \quad & \{\ \text{Converse of a Function}\ \} \\ & \mathit{roll} \cdot \mathsf{min}\ rr \cdot \Lambda(\mathit{foldrn}\ \mathit{add}\ \mathit{one}) \\ \supseteq \quad & \{\ \text{thinning}\ \} \\ & \mathit{roll} \cdot \mathsf{min}\ rr \cdot \mathsf{fold}\ (\mathsf{thin}\ q \cdot \Lambda((\mathit{one} \triangledown \mathit{add}) \cdot \mathsf{F}\ \mathit{choose})) \end{aligned}$$

Unfortunately, although the resulting algorithm appears to perform well in practice, its worst-case running time is exponential, and knowing this it is not too difficult to construct pathological inputs. The problem is that too few partial solutions are thinned out in the worst case.

4.5 An application of greediness

In fact, for Example 7 we can do a lot better. We have already observed that Theorem 3 does not apply; however, there is a connected preorder q which satisfies the conditions of Theorem 4. We choose the preorder q characterized by

$$\chi(q)\ (x, y) = \mathit{lspinecosts}\ x \trianglelefteq \mathit{lspinecosts}\ y$$

where $\trianglelefteq$ is the *lexicographic* (as opposed to pairwise) ordering on sequences: for sequences $as = [a_1, \ldots, a_m]$ and $bs = [b_1, \ldots, b_n]$, the ordering $as \trianglelefteq bs$ holds if and only if there exists an i with $0 \leq i \leq m, n$ such that $a_j = b_j$ for $1 \leq j \leq i$ and either $i = m$ (in which case as is a prefix of bs) or $i < m, n$ and $a_{i+1} < b_{i+1}$ (in which case as and bs differ first at position $i + 1$).

Again, $q \subseteq rr$, for the same reason that the q of §4.4 is included in rr. We leave the proof of monotonicity to Exercise 4.10.6, pausing only to observe as before that the costs of the reverse prefixes of the two spines of the trees constructed from $[1, 1, 1]$ are $[3, 1]$ and $[3, 2, 1]$, and the former is lexicographically (as well as pairwise) strictly less than the latter.

Now, however, q is connected, and so a greedy algorithm works — it suffices to keep a single partial solution at each stage. We have

$$\begin{array}{ll} & \mathsf{min}\ r \cdot \Lambda(\mathsf{fold}\ f) \\ \supseteq & \quad \{\ \mathsf{min}\ r \supseteq \mathsf{min}\ q\ \} \\ & \mathsf{min}\ q \cdot \Lambda(\mathsf{fold}\ f) \\ \supseteq & \quad \{\ \text{Refining Greedy Theorem, assuming } f \text{ monotonic under } q^\circ\ \} \\ & \mathsf{fold}\ (\mathsf{min}\ q \cdot \Lambda f) \end{array}$$

4.6 Refinement of the greedy algorithm to a program

Returning to bracketing problems in general, provided that we can find a connected preorder q under whose converse $one \triangledown add$ is monotonic (as we have seen we can do for the minimum height tree problem, for instance), we have

$$\begin{array}{ll} & \mathsf{min}\ r \cdot \Lambda(flatten^\circ) \\ = & \quad \{\ \text{Converse of a Function}\ \} \\ & roll \cdot \mathsf{min}\ rr \cdot \Lambda(foldrn\ add\ one) \\ \supseteq & \quad \{\ \text{strengthened preorder}\ \} \\ & roll \cdot \mathsf{min}\ q \cdot \Lambda(foldrn\ add\ one) \\ \supseteq & \quad \{\ \text{Greedy Theorem}\ \} \\ & roll \cdot foldrn\ (\mathsf{min}\ q \cdot \Lambda add)\ (\mathsf{min}\ q \cdot \Lambda one) \end{array}$$

To obtain a deterministic program, we still have to refine $\mathsf{min}\ q \cdot \Lambda one$ and $\mathsf{min} q \cdot \Lambda add$ to functions. Since one is a function, $\mathsf{min}\ q \cdot \Lambda one = one$. One can also show (Exercise 4.10.7) that $\mathsf{min}\ q \cdot \Lambda add \supseteq minadd$, where

$$minadd\ (a, (b, ts)) = (a, cons\ (roll\ (cons\ (b, us)), vs))$$

where $us +\!\!+ vs = ts$, and us is the shortest proper prefix of ts satisfying

$$max\ (a, cost\ (roll\ (cons\ (b, us)))) < cost\ (head\ vs)$$

If no such us exists, then $us = ts$ and $vs = [\,]$.

4.7 Summary

To summarize, the problem of building a tree of minimum height can be solved by computing

$$roll \cdot foldrp\ minadd\ one$$

Moreover, this algorithm takes linear time. The time taken to compute *minadd* is proportional to the length of the *us* chosen, but the length of the resulting spine is reduced by this amount; a standard amortization argument then shows that the total time taken for the algorithm is linear in the length of the given list.

4.8 The Haskell program

The program in Figure 1 is an implementation in Haskell [20] of the algorithm derived above for the minimum height tree instance of the bracketing problem. To avoid repeated computations, we label trees with their costs.

4.9 Bibliographic notes

The minimum height tree problem comes from [6], where a different greedy solution is presented. This chapter could be thought of as an abstract of Mu's forthcoming DPhil thesis [27], which expands on the approaches discussed here, in particular looking at other thinning algorithms, and exploring the relationship with dynamic programming as well as greedy algorithms.

4.10 Exercises

1. Prove the Converse-of-a-Function Theorem from §4.2.
2. Theorem 10 applies only to total functions f. If f is a partial function (that is, simple buyt not necessarily entire), the corresponding 'Converse of a Partial Function' theorem states, when $\mathsf{T} = \text{DATA}\ \mathsf{F}$, that $f^\circ = \mathsf{fold}_\mathsf{T}\ g$ provided that $f \cdot g \subseteq \mathsf{in} \cdot \mathsf{F}\ f$ and $\mathsf{dom}\ f = \mathsf{ran}\ g$. Prove this generalization.
3. Verify the application of the Converse-of-a-Function Theorem in §4.3.
4. Show that the Greedy Theorem is not applicable to (that is, $one \triangledown add$ is not monotonic under r° for) Example 9. What can you say about Example 8?
5. Show that for a bracketing problem $(\mathsf{A}, \oplus, \leq)$ such that $\oplus$ is commutative $(a \oplus b = b \oplus a)$, strict $(a \oplus b > a)$ and monotonic in its left argument $(a \leq a' \Rightarrow a \oplus b \leq a' \oplus b)$, the algebra $one \triangledown add$ is monotonic under q°, where q is as introduced in §4.4. Verify that these conditions apply for Examples 7 and 9.
6. Show that $one \triangledown add$ for the minimum height tree problem is monotonic under q°, where q is as introduced in §4.5.
7. Show that $\mathsf{min}\ q \cdot \Lambda add \supseteq minadd$ for the minimum height tree problem, where *minadd* is as defined in §4.6.
8. Express as an instance of the bracketing problem the problem of concatenating a list of lists into a single list in the cheapest possible way, when the cost of concatenating two lists is proportional to the length of the lefthand list. Derive a suitable algorithm for solving the problem.

```
data Tree  = Tip Int | Fork Int Tree Tree
type Spine = (Int, [Tree])

cost :: Tree -> Int
cost (Tip a)      = a
cost (Fork a t u) = a

fork :: Tree -> Tree -> Tree
fork t u = Fork (max (cost t) (cost u) + 1) t u

roll :: Spine -> Tree
roll (a, ts) = foldl fork (Tip a) ts

greedy :: [Int] -> Tree
greedy = roll . foldrp minadd one

one :: Int -> Spine
one a = (a, [])

minadd :: Int -> Spine -> Spine
minadd a (b,ts) = (a, split (Tip b : ts))
  where
    split [t] = [t]
    split (t:u:ts) = if   max a (cost t) < cost u
                     then t:u:ts
                     else split (fork t u : ts)

foldrp  ::  (a -> b -> b) -> (a -> b) -> ([a] -> b)
foldrp f g [x]    = g x
foldrp f g (x:xs) = f x (foldrp f g xs)
```

Fig. 1. Haskell implementation of minimum height tree algorithm

5 Bibliography

1. Roland Backhouse, Peter de Bruin, Grant Malcolm, Ed Voermans, and Jaap van der Woude. A relational theory of datatypes. In *STOP 1992 Summerschool on Constructive Algorithmics*. STOP project, 1992.
2. Roland Backhouse and Paul Hoogendijk. Elements of a relational theory of datatypes. In Bernhard Möller, Helmut Partsch, and Steve Schumann, editors, *LNCS 755: IFIP TC2/WG2.1 State-of-the-Art Report on Formal Program Development*, pages 7–42. Springer-Verlag, 1993.
3. R. Berghammer and H. Zierer. Relational algebraic semantics of deterministic and non-deterministic programs. *Theoretical Computer Science*, 43(2–3):123–147, 1986.
4. Richard Bird and Oege de Moor. Hybrid dynamic programming. Programming Research Group, Oxford, 1994.
5. Richard Bird and Oege de Moor. *The Algebra of Programming*. Prentice-Hall, 1996.
6. Richard S. Bird. On building trees with minimum height. *Journal of Functional Programming*, 7(4):441–445, 1997.
7. A. Bunkenburg. *Expression Refinement*. PhD thesis, Computing Science Department, University of Glasgow, 1997.
8. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
9. Sharon Curtis. *A Relational Approach to Optimization Problems*. PhD thesis, University of Oxford, 1996. Technical Monograph PRG-122.
10. J. W. de Bakker and W. P. de Roever. A calculus for recursive program schemes. In M. Nivat, editor, *Automata, Languages and Programming*, pages 167–196. North-Holland, 1973.
11. Oege de Moor and Jeremy Gibbons. Pointwise relational programming. In *LNCS 1816: Algebraic Methodology and Software Technology*, pages 371–390, May 2000.
12. S. Eilenberg and J. B. Wright. Automata in general algebras. *Information and Control*, 11(4):452–470, 1967.
13. Sharon Flynn. *A Refinement Calculus for Expressions*. PhD thesis, University of Glasgow, 1997.
14. P. J. Freyd and A. Ščedrov. *Categories, Allegories*. North-Holland, 1990.
15. David Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, 1981.
16. Eric C. R. Hehner. *A Practical Theory of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, 1993.
17. C. A. R. Hoare. Programs are predicates. In C. A. R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages*. Prentice-Hall, 1985. Also Chapter 20 of [18].
18. C. A. R. Hoare. *Essays in Computing Science*. Prentice Hall, 1989.
19. Paul Hoogendijk. *A Generic Theory of Datatypes*. PhD thesis, Technische Universiteit Eindhoven, 1997.
20. Simon Peyton Jones, John Hughes, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, and Philip Wadler. Haskell 98: A non-strict, purely functional language. `www.haskell.org/onlinereport`, February 1999.

21. Bernhard Korte, Laszlo Lovasz, and Rainer Schrader. *Greedoids*. Springer-Verlag, 1991.
22. Robert A. Kowalski. Predicate logic as a programming language. In *IFIP Congress*, 1974.
23. Ali Mili, Jules Desharnais, and Fatma Mili. *Computer Program Construction*. Oxford University Press, 1994.
24. Bernhard Möller. Relations as a program development language. In B. Möller, editor, *IFIP TC2/WG2.1 Working Conference on Constructing Programs from Specifications*, pages 373–397. North-Holland, 1991.
25. Joseph M. Morris. Programming by expression refinement: The KMP algorithm. In W. H. J. Feijen, A. J. M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is our Business*, chapter 37. Springer-Verlag, 1990.
26. Joseph M. Morris. Non-deterministic expressions and predicate transformers. *Information Processing Letters*, 61:241–246, 1997.
27. Shin-Cheng Mu. *Inverting Programs by Calculation*. DPhil thesis, University of Oxford, in preparation.
28. Theo Norvell and Eric Hehner. Logical specifications for functional programs. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *LNCS 669: Mathematics of Program Construction*, pages 269–290. Springer-Verlag, 1993.
29. Richard A. O'Keefe. *The Craft of Prolog*. MIT Press, 1990.
30. Alfred Tarski. On the calculus of relations. *Journal of Symbolic Logic*, 6(3):73–89, 1941.
31. Nigel Thomas Edgar Ward. *A Refinement Calculus for Nondeterministiuc Expressions*. PhD thesis, University of Queensland, February 1994.