

FUNCTIONAL PEARLS

Maximum Marking Problems

Richard S. Bird

*Programming Research Group, Oxford University
Wolfson Building, Parks Road, Oxford, OX1 3QD, UK*

1 Introduction

Here are two puzzles for you to solve. First, consider the binary tree in Figure 1. Take a pencil (I am assuming that this is your personal copy of JFP!) and mark some of the nodes in such a way that the sum of the values of marked nodes is as large as possible. The catch is that you cannot mark all the nodes: if you mark a node, then you are not allowed to mark its parent. Equivalently, no two marked nodes can be contiguous in the tree.

The second puzzle is similar though both the datatype and constraint are different. Consider the rose tree of Figure 2 (a rose tree is a tree with arbitrary branching structure). Mark some of the nodes so that all marked nodes are now *contiguous* in the tree. For example, if you mark the root value 4 and the leaf value 1, then you must also mark the values 5 and -3 along the path from 4 to 1. Again the idea is to maximise the sum of the marked nodes. Of course, if all values were nonnegative, the best solution would be to mark all nodes. But they aren't and the maximum sum is obtained only by a judicious choice of marking. Answers to both puzzles are given at the end of the paper.

The *maximum marking problem*, MMP for short, is the problem of marking the entries of some given data structure in such a way that a given constraint is satisfied and the sum of the values associated with marked entries is as large as possible. By the end of this pearl you will be convinced that there is a linear-time solution for both the puzzles described above.

Other variations of the MMP correspond to some well-known problems. If the

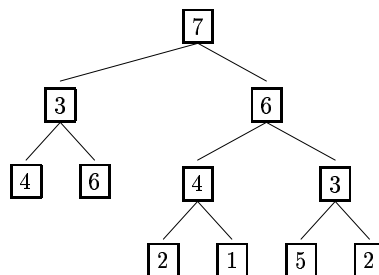


Fig. 1. A binary tree

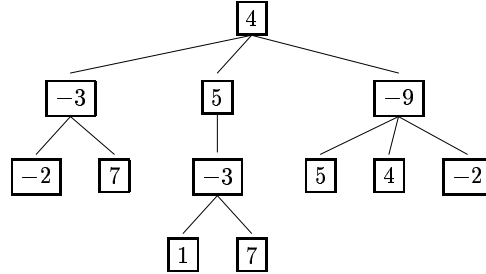


Fig. 2. A rose tree

data structure is a list of items along with their weights and values, and the marking constraint is that the sum of the weights of the items does not exceed a given quantity, then the problem of identifying the maximum value sum is just the classic Knapsack Problem. If the data structure is a list of numbers, and the constraint is that marked entries should be adjacent, then we have the well-known Maximum Segment Sum problem.

The theory behind these problems and how they can be solved efficiently is given in the chapter entitled “Thinning Algorithms” of (Bird and de Moor,1996). The real purpose of this pearl is to try and explain the essential ideas behind thinning algorithms in the context of a specific class of examples, without delving too much into the categorical theory of relations that forms the basis of (Bird and de Moor,1996). This pearl has also been written to try and answer the criticism made in (Sasano *et al.*,2000) to the effect that thinning algorithms are too difficult for functional programmers to apply in practice.

Let us end this introduction with one way of specifying the marking constraint of the first puzzle. We will mark a tree of type *Tree Int*, where

$$\mathbf{data} \text{ Tree } a = \text{Leaf } a \mid \text{Node } a (\text{Tree } a) (\text{Tree } a)$$

by attaching a Boolean label to each integer, where a *True* label indicates that an integer is marked and so contributes to the sum, while a *False* label indicates that it does not. The constraint is that the binary tree should be *atiguously* marked, a fancy name for a non-contiguous marking. One way to formalise atiguousness is to say that a marked tree x is atiguous if *atig* x returns a well-defined value, where

$$\begin{aligned} \text{atig} &:: \text{Tree } (\text{Int} \times \text{Bool}) \rightarrow \text{Bool} \\ \text{atig} &= \text{foldTree } \text{base } \text{step} \\ \text{base } (n, b) &= b \end{aligned}$$

and

$$\text{step } (n, b) \text{ } bx \text{ } by = \begin{cases} b \wedge \neg bx \wedge \neg by & \rightarrow \text{True} \\ \neg b & \rightarrow \text{False} \end{cases}$$

Thus, a tree x is atiguous iff *atig* x returns either *True* or *False*. The function *foldTree* is the fold function for the type *Tree a*. It is easy to see that *atig* x is undefined if, during the folding process and evaluation of *step* on tree x , a node is

encountered which is labelled *True* and one or both of its daughter nodes is also labelled *True*. Thus *atig* is a partial function. Haskell programmers can implement *atig* as a total function using *Maybe Bool* as the target type. The important point is that the target type of *atig* is finite. As we will see, the fact that *atig* can be expressed as a fold returning a value in a finite set is enough to guarantee a linear-time solution for the problem.

2 Specification

Forget binary trees, rose trees or lists, and imagine only that we are given a datatype $T\ Int$ of integers. The marking problem for T is solved by a function $mmp :: T\ Int \rightarrow Int$ that returns the maximum sum available. For simplicity we will concentrate only on the value of the best marking rather than on the marking itself, but it is just as easy to consider instead a function $mmp :: T\ Int \rightarrow T\ (Int \times Bool)$ that returns the best marking.

The function *mmp* can be specified in the following way:

$$mmp = \max(\leq) \cdot \Lambda(value \cdot dom\ test \cdot map_T\ mark)$$

The remainder of this section is devoted to explaining the notation and the subsidiary functions appearing in the specification of *mmp*.

First of all, the specification and its components are interpreted not as Haskell (or ML) functions over types living in the universe CPO_{\perp} of complete partial orders with bottom element \perp , but as *multifunctions* over types in the universe SET of ordinary sets. A multifunction is a nondeterministic function or, more simply, a *relation* that associates zero or more results with each argument. We indicate a multifunction by writing its type as $A \rightsquigarrow B$ rather than $A \rightarrow B$. For example,

$$\begin{aligned} mark &:: Int \rightsquigarrow Int \times Bool \\ mark\ n &= (n, True) \sqcap (n, False) \end{aligned}$$

The box \sqcap signifies nondeterministic choice. Thus *mark* is a multifunction that attaches an arbitrary Boolean value to an integer. To simplify subsequent type expressions, we introduce $Mark = Int \times Bool$, so $mark :: Int \rightsquigarrow Mark$.

The type of map_T is

$$map_T :: (a \rightsquigarrow b) \rightarrow (T\ a \rightsquigarrow T\ b)$$

This function, whose definition will be given a little later on, is just like the ordinary map function associated with a datatype T except that it can take a multifunction as argument and return a multifunction as result.

The combination $map_T\ mark$ denotes the operation of marking an element of $T\ Int$ in a completely nondeterministic way. The functional programmer may wonder at this point why multifunctions are being brought in. “Aren’t we going to be interested in the set of all possible markings?”, she may ask. The answer is: “Yes, but that will come later”. It is notationally much simpler to consider things at the level of an arbitrary marking, and then move wholesale to the set level at the last

possible moment. There is also another reason why multifunctions are necessary, as we will soon see.

Next, the function dom , which takes a multifunction as argument and returns a partial function as result, is defined as follows:

$$\begin{aligned} dom &:: (a \rightsquigarrow b) \rightarrow (a \rightsquigarrow a) \\ dom\ p &= fst \cdot \langle id, p \rangle \end{aligned}$$

The $split$ operation $\langle f, g \rangle$ is defined by $\langle f, g \rangle x = (f\ x, g\ x)$. The expression $\langle f, g \rangle$ denotes a multifunction that returns a result on an argument x if and only if both f and g do. We are working with types as flat sets remember, and there is no \perp or partially defined tuples. The function fst is standard and selects the first component of a (well-defined) pair. It will be appreciated from these remarks that $dom\ p$ applied to an argument x returns x if and only if x is in the *domain* of p . Thus, $dom\ p$ is a partial function included in the identity function. A partial function is a special case of a multifunction, namely a multifunction that returns either one value or none.

In the specification of mmp the function dom is applied to a given multifunction $test :: T\ Mark \rightsquigarrow Test$. For example, in the atiguous-marking problem, $test = atig$ and $Test = Bool$.

Next, the function $value$ is defined as follows:

$$\begin{aligned} value &:: T\ Mark \rightarrow Int \\ value &= sum \cdot map_T\ val \\ val\ (n, b) &= \mathbf{if\ } b \mathbf{\ then\ } n \mathbf{\ else\ } 0 \end{aligned}$$

The subsidiary function $sum :: T\ Int \rightarrow Int$ for summing a structure of integers will be defined shortly.

Next, the operation Λ turns a multifunction into the corresponding set-valued function:

$$\begin{aligned} \Lambda &:: (a \rightsquigarrow b) \rightarrow (a \rightarrow Set\ b) \\ (\Lambda f)\ a &= \{b \mid b \leftarrow f\ a\} \end{aligned}$$

We write $b \leftarrow f\ a$ to denote the fact that b is a possible value of $f\ a$. Thus $(\Lambda f)\ a$ returns the set of all possible values b that can be returned as the result of applying the multifunction f to a .

Finally, the multifunction max is defined by

$$\begin{aligned} max &:: (a \rightarrow a \rightarrow Bool) \rightarrow (Set\ a \rightsquigarrow a) \\ a \leftarrow max\ (\leq)\ as &\equiv a \in as \wedge (\forall b \in as : b \leq a) \end{aligned}$$

In words, max takes an ordering \leq and a set as as argument, and returns some maximum element in as under \leq . An ordering is also a relation, but it would seem strange to consider it as a multifunction, so we choose to represent it as a curried function returning a Boolean result. The minimum requirement on \leq is that it should be a *connected preorder*, that is, a reflexive and transitive relation with the property that for all x and y either $x \leq y$ or $y \leq x$. (The usual name for a connected preorder is a *total preorder* but that name invites confusion because of

the ambiguity of the word *total*.) Then we are guaranteed that $\max (\trianglelefteq) as$ produces at least one result for all nonempty sets as .

Note that $\max (\trianglelefteq) as$ does not specify which element of as should be chosen. This freedom of action is crucial to the success of the reasoning to come, and is the second reason why the move to multifunctions is necessary.

2.1 Folds and functors

Two operations, sum and map_T were left unspecified above (the multifunction $test$ is part of the input of the problem). To remedy the omission we need to say how T is defined and what the fold function for T is. In brief, a parameterised recursive datatype $T a$ can be defined as the least fixed point of another, so-called *base* datatype $F(a, b)$. The fixed-point property means that

$$T a \cong F(a, T a)$$

For example, $List a \cong 1 + a \times List a$, so $List a$ is a fixed point of $F(a, b) = 1 + a \times b$. The use of \cong rather than equality is because the two types are isomorphic rather than identical. The constructors of a **data** declaration in Haskell are functions that convert the component types on the right-hand side into elements of the type being declared. We can parcel all constructors into just one function $in_T : F(a, T a) \rightarrow T a$. The converse function $out_T : T a \rightarrow F(a, T a)$, the other half of the isomorphism, is implicit in the permitted use of pattern matching with elements of declared datatypes.

The least fixed-point property of T means that given any function $f :: F(a, b) \rightarrow b$ we can construct a unique function $h :: T a \rightarrow b$ satisfying

$$h \cdot in_T = f \cdot map_F(id, h)$$

The function h is denoted by $fold_F f$. Thus $fold_F :: (F(a, b) \rightarrow b) \rightarrow (T a \rightarrow b)$. For example, $sum = fold_F plus_F$, where $plus_F :: F(Int, Int) \rightarrow Int$. We cannot say what $plus_F$ is without knowing the structure of F . However, $plus_F$ can be defined for the so-called *regular* functors F (basically, polynomial functors closed under recursive **data** declarations) by induction over the structure of type constructors.

Given $f :: a \rightarrow x$ and $g :: b \rightarrow y$, the function $map_F(f, g)$ has type $F(a, b) \rightarrow F(x, y)$. Moreover, this function satisfies the two equations

$$\begin{aligned} map_F(id, id) &= id \\ map_F(f \cdot h, g \cdot k) &= map_F(f, g) \cdot map_F(h, k) \end{aligned}$$

In a word, F is a *functor*. We will make free use of the above equations in what follows without always making it explicit that we are doing so.

Given $f :: a \rightarrow b$, the combination $in_T \cdot map_F(f, id)$ has type $F(a, T b) \rightarrow T b$; consequently $fold_F(in_T \cdot map_F(f, id)) :: T a \rightarrow T b$. As might be suggested by this type signature, we have

$$map_T f = fold_F(in_T \cdot map_F(f, id))$$

Thus the action of T on functions is defined. Moreover, one can show that

$$\text{map}_T \text{id} = \text{id} \quad \text{and} \quad \text{map}_T (f \cdot g) = \text{map}_T f \cdot \text{map}_T g$$

Hence T is also a functor. The proof of this claim involves two facts that will be useful later on. The *identity* law states that $\text{fold}_F \text{in}_T$ is the identity function on $T a$. Thus,

$$\text{map}_T \text{id} = \text{fold}_F (\text{in}_T \cdot \text{map}_F (\text{id}, \text{id})) = \text{fold}_F \text{in}_T = \text{id}$$

and the first part is proved. The second law is called *type-functor fusion* and states that

$$\text{fold}_F f \cdot \text{map}_T g = \text{fold}_F (f \cdot \text{map}_F (g, \text{id}))$$

In words, a fold after a map can always be re-expressed as a single fold. We omit the simple proof that this gives the second claim.

All the above extends to the case that the argument of a fold is a multifunction rather than a plain function. There are one or two details that have to be addressed in a full explanation (such as what precisely does the functor \times mean in a relational setting), but we will not go into them. It suffices to state that we can take

$$\text{fold}_F \quad :: \quad (F(a, b) \rightsquigarrow b) \rightarrow (T a \rightsquigarrow b)$$

In particular, we can replace both occurrences of \rightsquigarrow by \rightarrow in this type signature.

2.2 The banana-split and fusion laws

We need two other pieces of technical machinery before we can proceed with deriving an implementation of *mmp*. The first is called the *banana-split* law, and states that

$$\langle \text{fold}_F f, \text{fold}_F g \rangle = \text{fold}_F \langle f \cdot \text{map}_F (\text{id}, \text{fst}), g \cdot \text{map}_F (\text{id}, \text{snd}) \rangle$$

In words, a split involving two folds can be rewritten as a fold involving a split. (In the old days, folds were written with “banana” brackets, hence the catchy name.)

The second piece of machinery is the *fusion* condition

$$f \cdot \text{fold}_F g \supseteq \text{fold}_F h \iff f \cdot g \supseteq h \cdot \text{map}_F (\text{id}, f)$$

The fusion condition also holds when both occurrences of \supseteq are replaced by $=$, or replaced by \subseteq . See (Bird,1998) for a discussion of the fundamental role of fusion in proving facts about functional programs. In fact, type-functor fusion is a special case of fusion. We will need fusion in Section 5.

3 Rewriting the specification

To save looking back, here is the specification of *mmp* again:

$$\text{mmp} = \text{max} (\leq) \cdot \Lambda(\text{value} \cdot \text{dom test} \cdot \text{map}_T \text{mark})$$

Let us start with the subexpression *value · dom test*:

$$\text{value} \cdot \text{dom test}$$

$$\begin{aligned}
&= \{ \text{definition of } \textit{dom} \} \\
&\quad \textit{value} \cdot \textit{fst} \cdot \langle \textit{id}, \textit{test} \rangle \\
&= \{ \text{claim} \} \\
&\quad \textit{fst} \cdot \langle \textit{value}, \textit{test} \rangle \\
&= \{ \text{definition of } \textit{value} \} \\
&\quad \textit{fst} \cdot \langle \textit{fold}_F \textit{ plus}_F \cdot \textit{map}_T \textit{ val}, \textit{test} \rangle \\
&= \{ \text{type functor fusion} \} \\
&\quad \textit{fst} \cdot \langle \textit{fold}_F (\textit{plus}_F \cdot \textit{map}_F (\textit{val}, \textit{id})), \textit{test} \rangle \\
&= \{ \text{assume } \textit{test} = \textit{fold}_F \textit{ part}_F \} \\
&\quad \textit{fst} \cdot \langle \textit{fold}_F (\textit{plus}_F \cdot \textit{map}_F (\textit{val}, \textit{id})), \textit{fold}_F \textit{ part}_F \rangle \\
&= \{ \text{banana split} \} \\
&\quad \textit{fst} \cdot \textit{fold}_F \textit{ f}
\end{aligned}$$

where, setting $\textit{Result} = \textit{Int} \times \textit{Test}$, we have

$$\begin{aligned}
\textit{f} &:: F(\textit{Mark}, \textit{Result}) \rightsquigarrow \textit{Result} \\
\textit{f} &= \langle \textit{plus}_F \cdot \textit{map}_F (\textit{val}, \textit{fst}), \textit{part}_F \cdot \textit{map}_F (\textit{id}, \textit{snd}) \rangle
\end{aligned}$$

The claim is a consequence of two laws:

$$\begin{aligned}
\textit{f} \cdot \textit{fst} &= \textit{fst} \cdot (\textit{f} \times \textit{id}) \\
(\textit{f} \times \textit{g}) \cdot \langle \textit{h}, \textit{k} \rangle &= \langle \textit{f} \cdot \textit{h}, \textit{g} \cdot \textit{k} \rangle
\end{aligned}$$

where $(\textit{f} \times \textit{g})(x, y) = (\textit{f} x, \textit{g} y)$. Note our assumption on $\textit{test} :: T \textit{Mark} \rightarrow \textit{Test}$, namely that $\textit{test} = \textit{fold}_F \textit{ part}_F$, where $\textit{part}_F :: F(\textit{Mark}, \textit{Test}) \rightsquigarrow \textit{Test}$.

Now we can continue:

$$\begin{aligned}
&\quad \textit{value} \cdot \textit{dom} \textit{ test} \cdot \textit{map}_T \textit{ mark} \\
&= \{ \text{above} \} \\
&\quad \textit{fst} \cdot \textit{fold}_F \textit{ f} \cdot \textit{map}_T \textit{ mark} \\
&= \{ \text{type functor fusion} \} \\
&\quad \textit{fst} \cdot \textit{fold}_F \textit{ g}
\end{aligned}$$

where $\textit{g} :: F(\textit{Int}, \textit{Result}) \rightsquigarrow \textit{Result}$ is defined by $\textit{g} = \textit{f} \cdot \textit{map}_F (\textit{mark}, \textit{id})$. Using this result, we now obtain

$$\begin{aligned}
&\quad \textit{max} (\leq) \cdot \Lambda(\textit{value} \cdot \textit{dom} \textit{ p} \cdot \textit{map}_T \textit{ mark}) \\
&= \{ \text{above} \} \\
&\quad \textit{max} (\leq) \cdot \Lambda(\textit{fst} \cdot \textit{fold}_F \textit{ g}) \\
&= \{ \text{claim} \} \\
&\quad \textit{fst} \cdot \textit{max} (\leq_1) \cdot \Lambda(\textit{fold}_F \textit{ g})
\end{aligned}$$

where $(\leq_1) :: \textit{Result} \rightarrow \textit{Result} \rightarrow \textit{Bool}$ is defined by

$$(\textit{a}_1, \textit{b}_1) \leq_1 (\textit{a}_2, \textit{b}_2) \hat{=} (\textit{a}_1 \leq \textit{a}_2)$$

The claim is intuitively obvious but to justify it calculationally would involve more notation than we want to expose. See (Bird and de Moor,1996) for the proof.

Where are we? Well, we have in effect reduced the MMP problem to one of computing an expression of the form $max(\sqsubseteq) \cdot \Lambda(fold_F h)$ efficiently. The only assumption was that *test* can be expressed as a fold. Time now for some standard theory.

4 Thinning

How can we compute an expression $max(\sqsubseteq) \cdot \Lambda(fold_F h)$, given suitable definitions of \sqsubseteq and h ? One possibility is to make use of the *Eilenberg-Wright* theorem, which says that

$$\Lambda(fold_F h) = fold_F(\Lambda(h \cdot map_F(id, choose)))$$

where $choose :: Set\ a \rightsquigarrow a$ is the membership relation for sets (so $a \leftarrow choose\ as$ iff $a \in as$). In words, 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 intermediate results. Proof of the Eilenberg-Wright theorem is given in (Bird and de Moor,1996), as are the proofs of other results in this section.

At the other extreme lies the *Greedy* theorem, which states that

$$max(\sqsubseteq) \cdot \Lambda(fold_F h) \supseteq fold_F(max(\sqsubseteq) \cdot \Lambda h)$$

provided $h :: F(a, b) \rightsquigarrow b$ is *monotonic* under the preorder $(\sqsubseteq) :: b \rightarrow b \rightarrow Bool$, that is,

$$x \sqsubseteq_F y \wedge u \leftarrow h\ x \Rightarrow (\exists v : v \leftarrow h\ y \wedge u \sqsubseteq v)$$

where $(\sqsubseteq_F) :: F(a, b) \rightarrow F(a, b) \rightarrow Bool$ is the ordering on F induced by \sqsubseteq . Note that the Greedy theorem asserts a *refinement* relation between the two sides, not an equality. In words, the Greedy theorem says that some optimum result (not necessarily every optimal result) can be computed by maintaining a single optimum partial result at each stage of the folding process.

For the MMP problem, $h :: F(Int, Result) \rightsquigarrow Result$ is given by

$$h = \langle plus_F \cdot map_F(val, fst), part_F \cdot map_F(id, snd) \rangle \cdot map_F(mark, id)$$

and it is not too difficult to see that h is *not* monotonic under \leq_1 : we can have $x \leq_1 y$ and $h\ x$ returning some result without having $h\ y$ returning any result because \leq_1 does not depend on second components. Hence monotonicity fails.

The minimum generalisation that restores monotonicity is to define \leq_2 by

$$(a_1, b_1) \leq_2 (a_2, b_2) \hat{=} (a_1 \leq a_2 \wedge b_1 = b_2)$$

Then it is fairly easy to see that h is monotonic under \leq_2 . The problem is that \leq_2 is not a connected preorder, so we cannot expect $max(\leq_2)\ as$ to return a value for all nonempty sets as .

What saves the day is the idea of *thinning*. Define

$$\begin{aligned} thin & :: (a \rightarrow a \rightarrow Bool) \rightarrow (Set\ a \rightsquigarrow Set\ a) \\ bs \leftarrow thin(\preceq)\ as & \hat{=} bs \subseteq as \wedge (\forall a \in as : \exists b \in bs : a \preceq b) \end{aligned}$$

In words, *thin* takes a not necessarily connected preorder \preceq and a set *as* and non-deterministically returns some subset *bs* of *as* with the property that all elements of *as* have an upper bound under \preceq in *bs*.

The thinning theorem states that

$$\max(\trianglelefteq) \cdot \Lambda(\text{fold}_F h) \supseteq \max(\trianglelefteq) \cdot \text{fold}_F k$$

where

$$\begin{aligned} k &:: F(a, \text{Set } b) \rightsquigarrow \text{Set } b \\ k &= \text{thin}(\trianglelefteq) \cdot \Lambda(h \cdot \text{map}_F(\text{id}, \text{choose})) \end{aligned}$$

provided that: (i) $x \preceq y \Rightarrow x \trianglelefteq y$; and (ii) *h* is monotonic under \preceq . In words, the thinning theorem says that we can compute an optimum result by maintaining a representative number of partial solutions at each stage of the folding process.

How many partial solutions have to be kept on the go for the MMP? Look again at the definition of \leq_2 (the instantiation for \preceq) and observe that the second components b_1 and b_2 are each elements of *Test*, the target type of *test*. If *Test* has size *k*, then we need keep at most *k* partial solutions at each stage. For the atiguous problem, *Test* = *Bool*, so only two partial solutions have to be kept. As we will see later on, *Test* will also have finite size for the other problem described in the introduction.

5 Implementation

As a consequence of applying the thinning theorem to the MMP problem we have

$$\begin{aligned} \text{mmp} &\supseteq \text{fst} \cdot \max(\leq_1) \cdot \text{fold}_F k \\ k &= \text{thin}(\leq_2) \cdot \Lambda(f \cdot \text{map}_F(\text{mark}, \text{choose})) \\ f &= \langle \text{plus}_F \cdot \text{map}_F(\text{val}, \text{fst}), \text{part}_F \cdot \text{map}_F(\text{id}, \text{snd}) \rangle \end{aligned}$$

The types of these multifunctions are as follows:

$$\begin{aligned} \text{mmp} &:: T \text{ Int} \rightarrow \text{Int} \\ k &:: F(\text{Int}, \text{Set Result}) \rightsquigarrow \text{Set Result} \\ f &:: F(\text{Mark}, \text{Result}) \rightsquigarrow \text{Result} \end{aligned}$$

Our task now is to implement the various functions in Haskell. In particular, *fold_F* can be implemented as in Figure 3 which uses Haskell's **newtype** construction to introduce the type *T*. The base functor *F* has to be supplied to complete the definition. Functions not decorated with an *F* subscript can be implemented independently of the details of any particular marking problem.

The aim of the game is to represent sets by lists and set-processing functions by list-processing ones. In particular, the functions *maxlist* :: *List a* → *a* and *thinlist* :: *List a* → *List a* are specified by

$$\begin{aligned} \max(\trianglelefteq) &\supseteq \text{maxlist}(\trianglelefteq) \cdot \text{listify} \\ \text{thin}(\trianglelefteq) &\supseteq \text{setify} \cdot \text{thinlist}(\trianglelefteq) \cdot \text{listify} \end{aligned}$$

where *setify* :: *List a* → *Set a* converts a list into the set of its elements and *listify* ::

```

newtype T a = InT (F a (T a))

foldF :: (F a b -> b) -> T a -> b
foldF f (InT x) = f (mapF id (foldF f) x)

```

Fig. 3. The datatype $T a$

$Set a \rightsquigarrow List a$ does the reverse. Thus

$$setify \cdot listify = id \quad \text{and} \quad listify \cdot setify \supseteq id$$

Use of \supseteq rather than $=$ is necessary in these formulae because we are replacing multifunctions by functions and hence exorcising nondeterminism. Figure 4 gives possible implementations of *maxlist* and *thinlist*.

It is important to note that *thinlist* works by comparing every element on the list with every other, and so takes quadratic time in the length of the list. Although *thinlist* is optimal at thinning, its quadratic behaviour is not acceptable for thinning algorithms in general. A better solution, explored in (Bird and de Moor,1996), is to implement a linear-time version of *thinlist* and combine it with a implementation of sets as *sorted* lists that bring candidates for thinning together, so allowing the linear-time version to be effective at thinning.

However, for an MMP problem that maintains k partial solutions at each step, *thinlist* will be applied to a list of length at most $2k$ at each step because application of *mark* doubles the number of candidates. Hence the quadratic definition of *thinlist* is acceptable, indeed welcome because it implies that we can implement a set by listing its elements in any order we like.

We have to implement Λ as a list-generating function. As a first step we “localize” occurrences of Λ by making use of three rules:

$$\begin{aligned}
\Lambda(f \cdot g) &= union \cdot map_S (\Lambda f) \cdot \Lambda g \\
\Lambda\langle f, g \rangle &= cp \cdot \langle \Lambda f, \Lambda g \rangle \\
\Lambda(map_F (f, g)) &= cp_F \cdot map_F (\Lambda f, \Lambda g)
\end{aligned}$$

The subsidiary functions have types

$$\begin{aligned}
union &:: Set (Set a) \rightarrow Set a \\
map_S &:: (a \rightarrow b) \rightarrow (Set a \rightarrow Set b) \\
cp &:: Set a \times Set b \rightarrow Set (a \times b) \\
cp_F &:: F (Set a, Set b) \rightarrow Set (F(a, b))
\end{aligned}$$

The function *union* takes the union of a set of sets, *map_S* is the map function for sets, *cp* takes the cartesian product of two sets, and *cp_F* is the generalisation of $cp = cp_{\times}$ to an arbitrary F .

If $g :: a \rightarrow b$, then $\Lambda g = \tau \cdot g$, where $\tau :: b \rightarrow Set b$ returns a singleton set. In such a case we have

$$\Lambda(f \cdot g) = \Lambda f \cdot g$$

Using these rules, together with the fact that $\Lambda choose$ is the identity function on

```

maxlist :: (a -> a -> Bool) -> [a] -> a
maxlist r = foldr1 max2 where max2 a b = if r a b then b else a

thinlist :: (a -> a -> Bool) -> [a] -> [a]
thinlist q = foldr step []
  where step a [] = [a]
        step a (b:x) | q a b = b:x
                      | q b a = a:x
                      | otherwise = b:step a x

cplist :: ([a],[b]) -> [(a,b)]
cplist (xs,ys) = [(x,y) | x <- xs, y <- ys]

tlist :: a -> [a]
tlist a = [a]

split :: (a -> b) -> (a -> c) -> a -> (b,c)
split f g x = (f x, g x)

type Mark = (Int,Bool)

mlist :: Int -> [Mark]
mlist n = [(n,True),(n,False)]

val :: Mark -> Int
val (a,b) = if b then a else 0

type Result = (Int,Test)

leq1 :: Result -> Result -> Bool
leq1 (a1,b1) (a2,b2) = (a1 <= a2)

leq2 :: Result -> Result -> Bool
leq2 (a1,b1) (a2,b2) = (a1 <= a2 && b1 == b2)

```

Fig. 4. Generic utility functions

sets, we can rewrite the specification of *mmp* to read:

$$\begin{aligned}
mmp &\supseteq fst \cdot max (\leq_1) \cdot fold_F k \\
k &= thin (\leq_2) \cdot union \cdot map_S f \cdot cp_F \cdot map_F (\Lambda mark, id) \\
f &= cp \cdot \langle \tau \cdot plus_F \cdot map_F (val, fst), \Lambda part_F \cdot map_F (id, snd) \rangle
\end{aligned}$$

The type of *f* is now $f :: F(\text{Mark}, \text{Result}) \rightarrow \text{Set Result}$.

Let us now implement *f*. We will need three functions *cplist*, *tlist* and *plist_F* satisfying

$$\begin{aligned}
cp \cdot (setify \times setify) &= setify \cdot cplist \\
\tau &= setify \cdot tlist \\
\Lambda part_F &= setify \cdot plist_F
\end{aligned}$$

```

mmp :: T Int -> Int
mmp = fst . maxlist leq1 . foldF klist
klist = thinlist leq2 . concat . map flist . cplistF . mapF mlist id
flist = cplist . split (tlist . plusF . mapF val fst) (plistF . mapF id snd)

```

Fig. 5. The generic program

Definitions of *cplist* and *tlist* are given in Figure 4; the definition of *plist_F* depends on the particular marking problem and will be given later. Installing these definitions we obtain $f = \text{setify} \cdot \text{flist}$, where *flist* is given in Figure 5.

Turning to *k*, we will need functions *mlist* and *cplist_F* satisfying

$$\begin{aligned} \Lambda \text{mark} &= \text{setify} \cdot \text{mlist} \\ \text{cp}_F \cdot \text{map}_F (\text{setify}, \text{setify}) &= \text{setify} \cdot \text{cplist}_F \end{aligned}$$

Figure 4 gives one implementation of *mlist*; the definition of *cplist_F* depends on *F* and will be given later. Installing these identities and using the rules relating *listify* and *setify* given above, together with the equation

$$\text{union} \cdot \text{map}_S \text{setify} \cdot \text{setify} = \text{setify} \cdot \text{concat}$$

we obtain

$$\text{listify} \cdot k \supseteq \text{klist} \cdot \text{map}_F (\text{id}, \text{listify})$$

where *klist* is defined in Figure 5.

Finally, we turn to *mmp* and reason:

$$\begin{aligned} &\text{mmp} \\ \supseteq &\quad \{\text{given}\} \\ &\text{fst} \cdot \text{max} (\leq_1) \cdot \text{fold}_F k \\ \supseteq &\quad \{\text{specification of } \text{maxlist}\} \\ &\text{fst} \cdot \text{maxlist} (\leq_1) \cdot \text{listify} \cdot \text{fold}_F k \\ \supseteq &\quad \{\text{fusion; see above}\} \\ &\text{fst} \cdot \text{maxlist} (\leq_1) \cdot \text{fold}_F \text{klist} \end{aligned}$$

The result as a Haskell program is summarised in Figure 5.

6 Applications

Finally we are in a position to instantiate the generic marking problem for our two examples. Figure 6 gives the instantiations of the remaining functions *map_F*, *cplist_F*, *plus_F* and *plist_F* for the atiguous problem on binary trees.

In order to bring out a crucial point about efficiency, we have declared the type of *cplist_F* to be the instance at which it is used in the generic marking problem. If *Result* has size *k* (and $k = 2$ in the atiguous problem), then the thinning algorithm maintains *k* partial results at each step. Consequently, *cplist_F* generates $2k^2$

```

data F a b = Leaf a | Fork a b b
type Test = Bool

mapF :: (a -> c) -> (b -> d) -> F a b -> F c d
mapF f g (Leaf a)      = Leaf (f a)
mapF f g (Fork a b1 b2) = Fork (f a) (g b1) (g b2)

cplistF :: F [Mark] [Result] -> [F Mark Result]
cplistF (Leaf as) = [Leaf a | a <- as]
cplistF (Fork as bs1 bs2)
  = [Fork a b1 b2 | a <- as, b1 <- bs1, b2 <- bs2]

plusF :: F Int Int -> Int
plusF (Leaf a)      = a
plusF (Fork a b1 b2) = a + b1 + b2

plistF :: F Mark Test -> [Test]
plistF (Leaf (a,b)) = [b]
plistF (Fork (a,b) b1 b2)
  | b && not b1 && not b2 = [True]
  | not b                = [False]
  | otherwise           = []

```

Fig. 6. The atiguous problem

candidate new partial results for subsequent thinning. These partial results can be processed in constant time, so the complete algorithm requires linear time.

The situation changes when the base functor F is not polynomial, as is the case with rose trees. For rose trees we have $F(a, b) = a \times List\ b$, and for a node with n immediate offspring, the associated function $cplist_F$ will produce a list of $2k^n$ candidates. Although these candidates are subsequently thinned to only k results, the process will no longer take constant time. In fact, direct instantiation of any non-trivial marking problem for rose trees will take exponential time.

The only way out of this unfortunate situation seems to be to recast a marking problem for rose trees as a marking problem for leaf-labelled binary trees, exploiting the isomorphism between rose trees and binary trees illustrated in Figure 7. This is also the resolution proposed by (Sasano *et al.*, 2000) though the problem is not identified in quite the same way.

Figure 8 gives the instantiation for the contiguous problem for rose trees, expressed as a problem on binary trees. The type *Test* contains three values: *A* signifies that the tree is contiguous and the root is marked; *B* that the tree is contiguous and the root is unmarked (that is, marked *False*), and *C* that the tree is completely unmarked. The definition of $plist_F$ is reasonably clear once one ploughs through the clauses.

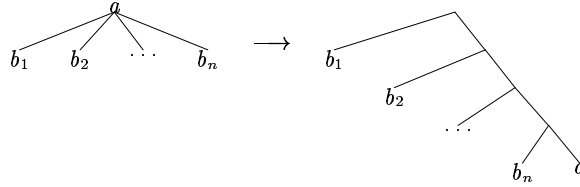


Fig. 7. Isomorphism between rose trees and leaf-labelled binary trees

```

data F a b = Leaf a | Fork b b
data Test = A | B | C deriving (Eq,Ord)

mapF :: (a -> c) -> (b -> d) -> F a b -> F c d
mapF f g (Leaf a)      = Leaf (f a)
mapF f g (Fork b1 b2) = Fork (g b1) (g b2)

cplistF :: F [a] [b] -> [F a b]
cplistF (Leaf as)      = [Leaf a | a <- as]
cplistF (Fork bs1 bs2) = [Fork b1 b2 | b1 <- bs1, b2 <- bs2]

plusF :: F Int Int -> Int
plusF (Leaf a)      = a
plusF (Fork b c) = b+c

plistF :: F Mark Test -> [Test]
plistF (Leaf (a,b)) = if b then [A] else [C]
plistF (Fork A A)   = [A]
plistF (Fork C A)   = [A]
plistF (Fork A C)   = [B]
plistF (Fork B C)   = [B]
plistF (Fork C B)   = [B]
plistF (Fork C C)   = [C]
plistF (Fork x y)   = []

```

Fig. 8. The contiguous problem

7 Conclusions

Consider the benefits of the treatment above: we have derived a generic solution to *all* marking problems whose constraint can be expressed as a fold involving a multi-function. The only fly in the ointment is that, for efficiency, any problem involving a datatype that is not based on a polynomial functor has to be re-expressed in terms of a datatype that is. What is more, we have shown how thinning algorithms can solve a whole range of optimisation problems. (Oh, yes, the two puzzles yield maximum values 28 and 18 respectively).

References

- Bird, R. (1998) *Introduction to Functional Programming using Haskell*, Prentice Hall International.
- Bird, R. and de Moor, O. (1998) *The Algebra of Programming*, Prentice Hall International.
- Sasano, I, Hu, Z, Takeichi, M, and Ogawa, M. (2000) Calculating linear-time algorithms for solving maximum weightsum problems. *Proc International Conference on Functional Programming*, Montreal, Canada.