J. Functional Programming **11** (4): 411–424, July 2001. Printed in the United Kingdom

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) 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 data structure is a list of items along with their weights and values, and the marking

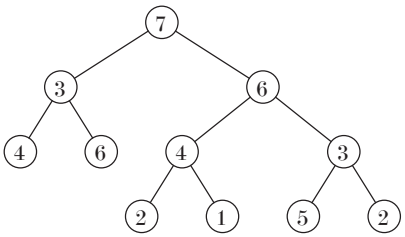

Fig. 1. A binary tree.

https://doi.org/10.1017/S0956796801004038 Published online by Cambridge University Press

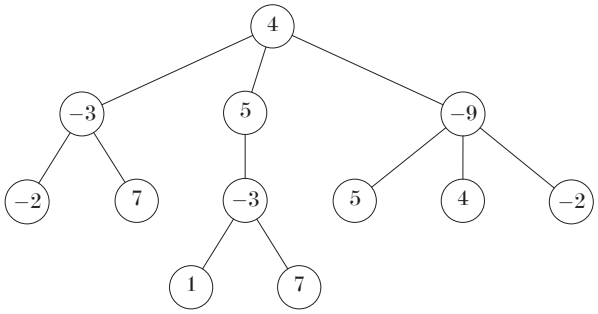

Fig. 2. A rose tree.

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 progammers 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

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

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

$$\begin{array}{lcl} atig & :: & Tree\ (Int \times Bool) \to Bool \\ atig & = & foldTree\ base\ step \\ base\ (n, b) & = & b \end{array}$$

and

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

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

https://doi.org/10.1017/S0956796801004038 Published online by Cambridge University Press

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 $mmp :: T\ Int \to Int$ 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 $mmp :: T\ Int \to T\ (Int \times Bool)$ that returns the best marking.

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

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

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 $A \to B$. For example,

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

The box $\Box$ 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

$$map_T \quad :: \quad (a \rightsquigarrow b) \to (T\ a \rightsquigarrow T\ b)$$

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

https://doi.org/10.1017/S0956796801004038 Published online by Cambridge University Press

partial function as result, is defined as follows:

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

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:

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

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:

$$\begin{array}{lcl} \Lambda & :: & (a \rightsquigarrow b) \rightarrow (a \rightarrow Set\ b) \\ (\Lambda f)\, a & = & \{b \mid b \leftsquigarrow f\ a\} \end{array}$$

We write $b \leftsquigarrow f\ a$ 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

$$\begin{array}{lcl} max & :: & (a \rightarrow a \rightarrow Bool) \rightarrow (Set\ a \rightsquigarrow a) \\ a \leftarrow max\ (\trianglelefteq)\, as & \equiv & a \in as\ \wedge\ (\forall b \in as : b \trianglelefteq a) \end{array}$$

In words, *max* takes an ordering $\trianglelefteq$ and a set *as* as argument, and returns some maximum element in *as* under $\trianglelefteq$. 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 $\trianglelefteq$ 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 $x \trianglelefteq y$ or $y \trianglelefteq x$. (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.

https://doi.org/10.1017/S0956796801004038 Published online by Cambridge University Press

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

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

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 $in_T : F(a, T\,a) \to T\,a$. The converse function $out_T : T\,a \to F(a, T\,a)$, 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 $f :: F(a,b) \to b$ we can construct a unique function $h :: T\,a \to b$ satisfying

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

The function h is denoted by $fold_F\,f$. Thus $fold_F :: (F(a,b) \to b) \to (T\,a \to b)$. For example, $sum = fold_F\,plus_F$, where $plus_F :: F(Int, Int) \to Int$. 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 $f :: a \to x$ and $g :: b \to y$, the function $map_F\,(f,g)$ has type $F(a,b) \to F(x,y)$. 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 $f :: a \to b$, the combination $in_T \cdot map_F\,(f, id)$ has type $F(a, T\,b) \to T\,b$; consequently $fold_F\,(in_T \cdot map_F\,(f, id)) :: T\,a \to T\,b$. As might be suggested by this type signature, we have

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

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

$$map_T\,id = id \quad \text{and} \quad map_T\,(f \cdot g) = map_T\,f \cdot map_T\,g$$

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 $fold_F\,in_T$ is the identity function on $T\,a$. Thus,

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

https://doi.org/10.1017/S0956796801004038 Published online by Cambridge University Press

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

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

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

$$fold_F \quad :: \quad (F(a,b) \rightsquigarrow b) \rightarrow (T\, a \rightsquigarrow b)$$

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

$$\langle fold_F\, f, fold_F\, g \rangle \quad = \quad fold_F\, \langle f \cdot map_F\, (id, fst), g \cdot map_F\, (id, snd) \rangle$$

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

$$f \cdot fold_F\, g \sqsupseteq fold_F\, h \quad \Leftarrow \quad f \cdot g \sqsupseteq h \cdot map_F\, (id, f)$$

The fusion condition also holds when both occurrences of $\sqsupseteq$ 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:

$$mmp \quad = \quad max\, (\leqslant) \cdot \Lambda(value \cdot dom\, test \cdot map_T\, mark)$$

Let us start with the subexpression $value \cdot dom\, test$:

$$\begin{array}{ll} & value \cdot dom\, test \\ = & \quad \{\text{definition of } dom\} \\ & value \cdot fst \cdot \langle id, test \rangle \\ = & \quad \{\text{claim}\} \\ & fst \cdot \langle value, test \rangle \\ = & \quad \{\text{definition of } value\} \\ & fst \cdot \langle fold_F\, plus_F \cdot map_T\, val, test \rangle \end{array}$$

https://doi.org/10.1017/S0956796801004038 Published online by Cambridge University Press

$$
\begin{array}{ll}
= & \{\text{type functor fusion}\} \\
& fst \cdot \langle fold_F\,(plus_F \cdot map_F\,(val, id)), test\rangle \\
= & \{\textbf{assume}\ test = fold_F\ part_F\} \\
& fst \cdot \langle fold_F\,(plus_F \cdot map_F\,(val, id)), fold_F\ part_F\rangle \\
= & \{\text{banana split}\} \\
& fst \cdot fold_F\, f
\end{array}
$$

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

$$
\begin{array}{lcl}
f & :: & F(Mark, Result) \rightsquigarrow Result \\
f & = & \langle plus_F \cdot map_F\,(val, fst), part_F \cdot map_F\,(id, snd)\rangle
\end{array}
$$

The claim is a consequence of two laws:

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

where $(f \times g)(x, y) = (f\,x, g\,y)$. Note our assumption on $test :: T\,Mark \to Test$, namely that $test = fold_F\ part_F$, where $part_F :: F(Mark, Test) \rightsquigarrow Test$.

Now we can continue:

$$
\begin{array}{ll}
& value \cdot dom\ test \cdot map_T\ mark \\
= & \{\text{above}\} \\
& fst \cdot fold_F\, f \cdot map_T\ mark \\
= & \{\text{type functor fusion}\} \\
& fst \cdot fold_F\ g
\end{array}
$$

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

$$
\begin{array}{ll}
& max\ (\leqslant) \cdot \Lambda(value \cdot dom\ p \cdot map_T\ mark) \\
= & \{\text{above}\} \\
& max\ (\leqslant) \cdot \Lambda(fst \cdot fold_F\ g) \\
= & \{\text{claim}\} \\
& fst \cdot max\ (\leqslant_1) \cdot \Lambda(fold_F\ g)
\end{array}
$$

where $(\leqslant_1) :: Result \to Result \to Bool$ is defined by

$$(a_1, b_1) \leqslant_1 (a_2, b_2) \quad \widehat{=} \quad (a_1 \leqslant a_2)$$

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 $max\ (\trianglelefteq) \cdot \Lambda(fold_F\ h)$ efficiently. The only assumption was that *test* can be expressed as a fold. Time now for some standard theory.

https://doi.org/10.1017/S0956796801004038 Published online by Cambridge University Press

4 Thinning

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

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

where $choose :: Set\ a \rightsquigarrow a$ is the membership relation for sets (so $a \leftsquigarrow choose\ as$ 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

$$max\ (\trianglelefteq) \cdot \Lambda(fold_F\ h) \quad \supseteq \quad fold_F\ (max\ (\trianglelefteq) \cdot \Lambda h)$$

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

$$x \trianglelefteq_F y \ \wedge\ u \leftsquigarrow h\ x \quad \Rightarrow \quad (\exists v : v \leftsquigarrow h\ y \wedge u \trianglelefteq v)$$

where $(\trianglelefteq_F) :: F(a,b) \to F(a,b) \to Bool$ is the ordering on F induced by $\trianglelefteq$. 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

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

and it is not too difficult to see that h is *not* monotonic under $\leqslant_1$: we can have $x \leqslant_{1F} y$ and $h\ x$ returning some result without having $h\ y$ returning any result because $\leqslant_1$ does not depend on second components. Hence monotonicity fails.

The minimum generalisation that restores monotonicity is to define $\leqslant_2$ by

$$(a_1, b_1) \leqslant_2 (a_2, b_2) \quad \widehat{=} \quad (a_1 \leqslant a_2 \ \wedge\ b_1 = b_2)$$

Then it is fairly easy to see that h is monotonic under $\leqslant_2$. The problem is that $\leqslant_2$ is not a connected preorder, so we cannot expect $max\ (\leqslant_2)\ as$ to return a value for all nonempty sets as.

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

$$\begin{array}{lcl} thin & :: & (a \to a \to Bool) \to (Set\ a \rightsquigarrow Set\ a) \\ bs \leftarrow thin\ (\preceq)\ as & \equiv & bs \subseteq as \ \wedge\ (\forall a \in as : \exists b \in bs : a \preceq b) \end{array}$$

In words, *thin* takes a not necessarily connected preorder $\preceq$ and a set as and nondeterministically 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

$$max\ (\trianglelefteq) \cdot \Lambda(fold_F\ h) \quad \supseteq \quad max\ (\trianglelefteq) \cdot fold_F\ k$$

https://doi.org/10.1017/S0956796801004038 Published online by Cambridge University Press

```
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 $T\,a$.

where

$$\begin{array}{lcl} k & :: & F(a, Set\ b) \rightsquigarrow Set\ b \\ k & = & thin\,(\preceq) \cdot \Lambda\,(h \cdot map_F\,(id, choose)) \end{array}$$

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 $\leqslant_2$ (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

$$\begin{array}{lcl} mmp & \supseteq & fst \cdot max\,(\leqslant_1) \cdot fold_F\,k \\ k & = & thin\,(\leqslant_2) \cdot \Lambda(f \cdot map_F\,(mark, choose)) \\ f & = & \langle plus_F \cdot map_F\,(val, fst), part_F \cdot map_F\,(id, snd)\rangle \end{array}$$

The types of these multifunctions are as follows:

$$\begin{array}{lcl} mmp & :: & T\ Int \to Int \\ k & :: & F(Int, Set\ Result) \rightsquigarrow Set\ Result \\ f & :: & F(Mark, Result) \rightsquigarrow Result \end{array}$$

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 $maxlist :: List\ a \to a$ and $thinlist :: List\ a \to List\ a$ are specified by

$$\begin{array}{lcl} max\,(\trianglelefteq) & \supseteq & maxlist\,(\trianglelefteq) \cdot listify \\ thin\,(\trianglelefteq) & \supseteq & setify \cdot thinlist\,(\trianglelefteq) \cdot listify \end{array}$$

where $setify :: List\ a \to Set\ a$ converts a list into the set of its elements and

https://doi.org/10.1017/S0956796801004038 Published online by Cambridge University Press

```
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.

listify $:: Set\ a \rightsquigarrow List\ a$ 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

https://doi.org/10.1017/S0956796801004038 Published online by Cambridge University Press

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:

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

The subsidiary functions have types

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

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

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

Using these rules, together with the fact that $\Lambda choose$ is the identity function on sets, we can rewrite the specification of *mmp* to read:

$$\begin{array}{lcl} mmp & \supseteq & fst \cdot max\,(\leqslant_1) \cdot fold_F\,k \\ k & = & thin\,(\leqslant_2) \cdot union \cdot map_S\,f \cdot cp_F \cdot map_F\,(\Lambda mark, id)) \\ f & = & cp \cdot \langle \tau \cdot plus_F \cdot map_F\,(val, fst), \Lambda part_F \cdot map_F\,(id, snd)\rangle \end{array}$$

The type of f is now $f :: F(Mark, Result) \to Set\,Result$.

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

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

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

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

$$\begin{array}{lcl} \Lambda mark & = & setify \cdot mlist \\ cp_F \cdot map_F\,(setify, setify) & = & setify \cdot cplist_F \end{array}$$

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*

https://doi.org/10.1017/S0956796801004038 Published online by Cambridge University Press

```
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.

and *setify* given above, together with the equation

$$union \cdot map_S\ setify \cdot setify \quad = \quad setify \cdot concat$$

we obtain

$$listify \cdot k \quad \supseteq \quad klist \cdot map_F\ (id, listify)$$

where *klist* is defined in Figure 5.

Finally, we turn to *mmp* and reason:

$$\begin{array}{ll} & mmp \\ \supseteq & \quad \{\text{given}\} \\ & fst \cdot max\ (\leqslant_1) \cdot fold_F\ k \\ \supseteq & \quad \{\text{specification of } maxlist\} \\ & fst \cdot maxlist\ (\leqslant_1) \cdot listify \cdot fold_F\ k \\ \supseteq & \quad \{\text{fusion; see above}\} \\ & fst \cdot maxlist\ (\leqslant_1) \cdot fold_F\ klist \end{array}$$

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.

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 $2k^2$ 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

https://doi.org/10.1017/S0956796801004038 Published online by Cambridge University Press

```
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.

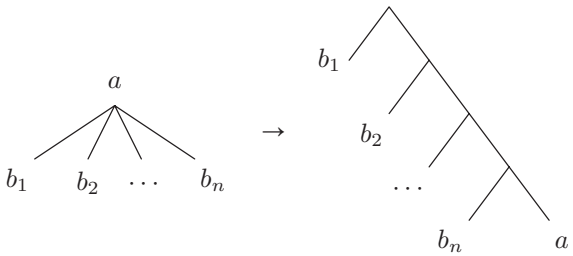

Fig. 7. Isomorphism between rose trees and leaf-labelled binary trees.

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 (i.e. marked *False*), and *C* that the tree is completely unmarked. The definition of $plist_F$ is reasonably clear once one ploughs through the clauses.

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

https://doi.org/10.1017/S0956796801004038 Published online by Cambridge University Press

```
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.

a multifunction. 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.

Bird, R. and de Moor, O. (1998) *The Algebra of Programming*. Prentice Hall.

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.

https://doi.org/10.1017/S0956796801004038 Published online by Cambridge University Press