Some optimisation problems revisited

Henry B. McLoughlin[†]

Department of Computer Science, University College Dublin, Belfield, Dublin 4.

Kevin Hely

Department of Computer Science, University College Dublin, Belfield, Dublin 4.

Abstract

We consider some simple optimisation problems and employ a non-traditional method to solve them. We try to model both the problem and solution domains as algebraic structures, attempting to characterise the join operations on these domains. In each of the examples chosen, these structures turn out to be monoids and we make use of the homomorphisms between the problem and solution structures to construct recursive programs. We argue that this method employs a particular conceptual model which is easy to use and may produce nice results.

1 Introduction

Monoids are rather simple algebraic structures which have proved useful in modelling some of the structures that are of interest to us as computer scientists. In addition, many of the structure transformations which we express as algorithms have their counterpart in a variety of morphisms between monoids.

Mac an Airchinnigh, originally in his doctoral thesis [6] and in subsequent papers [7, 8], advocates the use of monoids and their homomorphisms both in specification and in program development. In his early work, he appears to be motivated partly by a desire to provide a sound mathematical basis for these activities but also out of pedagogic concerns; arguing that equipping students with an appropriate elegant conceptual model enhances their ability to analyse problems, articulate their properties and develop appropriate solutions.

Subsequently his work has moved into examining richer algebraic structures although we believe he still retains the same motivations. What we describe in this paper is some of our initial efforts which are in the spirit of his work.

In earlier work, we took the techniques of Dijkstra and others [2, 3, 4, 5] on the rigorous development of imperative programs from formal specifications and attempted to explicitly capture a variety of solution tactics into general problem-solving templates. A template comprises a general program specification and its corresponding solution. A particular problem may be solved by comparing its specification to those of the templates, choosing one that matches it, and instantiating the template with the parameters of the problem to obtain a correct solution. This approach may be likened to the technique of differentiation by rule in analysis where, instead of calculating the derivative of a given function from first principles, we simply apply one of a set of rules to obtain the derivative directly [9].

This gives rise to problem classifications such as partitioned and non-partitioned sequence processing, linear and bounded linear searches, slope searches and binary chops to mention a few. We have described some of these and shown how they are applied in an earlier paper [9].

This approach has proved effective with beginning students learning to develop correct programs from specifications. It has also given us, as curriculum developers, a useful language in which to describe the programming skill levels which students should attain from particular courses.

Now we want to do the same for functional programming. It is our belief that identifying the join operations over the algebraic structures which are the subjects of our computations will have a key role in the development of functional programs; much the same as identifying appropriate invariants plays a key role in developing loop programs. This paper describes some initial efforts at taking this approach with some familiar and quite easy optimisation problems.

We surmise that an optimal solution of size k say, can be found by composing optimal solutions to problems of sizes i and j (where $i+j=k$). In particular, we hope to compose solutions of sizes 1 and $k-1$. Optimal solutions to problems of size 1 are generally easy to compute and if we can characterise the composition operator for particular problems then we can develop recursive solutions to problems which have a worst-case complexity $O(N \times \xi)$ where ξ denotes the worst-case complexity of the composition operator and N is the size of the problem instance.

Moreover, if we are examining the set of optimal solutions for different instances of a particular problem

[†] Corresponding author.

and if this is closed under a suitable composition operator then (allowing for the existence of the null or empty solution) we are studying *monoids of optimal solutions*.

In section 2 we consider finding the optimal strategy for merging a set of sorted files so as to minimise the overall cost of the merge operation. We show that there exists a monoid of optimal merge strategies and use this to construct the solution.

Following on from this, in section 3 we apply this approach to a problem about representing amounts of money using the smallest number of coins of particular denominations. We start with a familiar example for which there exists a trivial solution, but choose to ignore its obvious solution. We justify that choice by considering a less familiar example and showing how our approach may be used in its solution.

Finally in section 4, we make some observations on this preliminary work and indicate some future directions for research.

2 A file merge problem

Suppose we are given a set of files $F_0, \dots, F_{n-1}$, of lengths $L_0, \dots, L_{n-1}$ respectively, each of which is sorted according to some ordering. Suppose in addition we have a merge operator $:$ which maps files F_i and F_j to a sorted file of length $L_i + L_j$ which comprises the contents of files F_i and F_j . The efficiency of such an operator is given to be proportional to the sum of the lengths of the files merged. (For the sake of simplicity, we take the constant of proportionality to be 1.)

We are asked to develop a strategy for merging the set of files that minimises the cumulative cost of the merges.

2.1 Variations in the costs can occur

Consider a very simple example. Suppose we have three files F_0, F_1 and F_2 of lengths 5, 15 and 20, respectively. Given that merging is commutative, we can reduce the number of possible strategies for merging the files to three cases. These are illustrated below where the order of merging is indicated by appropriate parenthesisation.

Case	Merge	Cost
(i)	$F_0 : (F_1 : F_2)$	$(5+(15+20))+(15+20) = 75$
(ii)	$(F_0 : F_1) : F_2$	$(5+15)+((5+15)+20) = 60$
(iii)	$(F_0 : F_2) : F_1$	$(5+20)+((5+20)+15) = 65$

Clearly the strategy employed has a significant bearing on the resulting cost.

2.2 Representing the merge strategy

We choose to represent the merge strategy as a type of binary tree. The values of the leaf nodes represent the lengths of the given files. The value at any non-leaf node represents the length of the file that results from merging the files whose lengths are the left and right sub-trees of the particular node. We refer to this structure as a binary merge tree.

Definition. A *binary merge tree* (BMT) is an object of the type

$$BMT == \text{empty} \mid \text{node } BMT \mathbf{N} BMT$$

We make use of the (postfix) function *val* defined as follows:

$$\begin{aligned} \text{empty.val} &= 0 \\ (\text{node } \alpha \ b \ \gamma).\text{val} &= b \end{aligned}$$

and the predicates *Is-empty* and *Is-leaf*, defined as follows:

$$\begin{aligned} \text{Is-empty}(\text{empty}) & \\ \neg \text{Is-empty}(\text{node } \alpha \ b \ \gamma) & \\ \text{Is-leaf}(\text{node } \alpha \ b \ \gamma) &\equiv \alpha = \gamma = \text{empty}. \end{aligned}$$

The BMTs corresponding to the strategies in the previous section may be illustrated as in Figure 1.

Figure 1: BMTs corresponding to cases (i)–(iii).

2.3 The cost of a BMT

We would like to compare the costs of strategies represented by their corresponding binary merge trees. The cost of the strategy represented by a BMT is given by the following function:

$$\begin{aligned} \text{Cost}(\text{empty}) &= 0 \\ \text{Cost}(\text{node } \alpha \text{ } b \text{ } \gamma) &= b + \text{Cost}(\alpha) + \text{Cost}(\gamma) \end{aligned}$$

2.4 The existence of optimal solutions

For a given set of files there exists a set of BMTs representing all possible strategies for merging those files. This set can be ordered using the cost function given above.

As an aside, note that since the merge operator is commutative, a number of BMTs in a given set will have the same cost. In addition, there may exist a number of distinct BMTs representing distinct strategies which also give rise to the same cost.

The cost ordering is a partial ordering. Therefore there exists a greatest lower bound which corresponds to (one of the) minimum cost BMT(s) for the given set of files. This BMT represents the optimal merge strategy.

For two sets of files A and B we have the corresponding sets T_A and T_B of their BMTs. There are many join operators $\oplus_i$ on BMT pairs $(\alpha, \beta) \in T_A \times T_B$ each of which yields a set S_i of BMTs over $A \cup B$. The union ($\cup_i: S_i$) of these sets is the set of *all possible* BMTs over $A \cup B$. This set can be ordered using the cost function giving rise to a greatest lower bound corresponding to the optimal merge strategy for the set of files $A \cup B$.

However, we only wish to consider whether there exists a join operator which, given any two *minimum cost* BMTs over A and B respectively, yields a minimum cost BMT over $A \cup B$. Ideally the empty BMT (i.e. the BMT corresponding to a merge of zero files) will be its identity. The reader will note that we aim to demonstrate the existence of the monoid of minimum cost BMTs.

Parenthetically we mention that monoids such as this —that preserve the minimal nature of some quantity— are quite common. For example, given a volume, there are many geometric shapes containing it. From geometry we know that a sphere is the structure of minimum surface area containing a given volume. The volume of a sphere of radius $r \in \mathbf{R}^+$ is given by $(4\pi r^3)/3$. Therefore, the monoid $(\mathbf{R}^+, :, 0)$ can be interpreted as the additive monoid of volumes which preserves the minimum surface area attribute, where $:$ is defined as follows:

$$r : s = \sqrt[3]{r^3 + s^3}$$

Both r and s are member of the set $\mathbf{R}^+$ and are interpreted as the radii of the two spheres whose volumes are to be composed.

2.5 The monoid of minimum cost BMTs

Let M_F denote the set of minimum cost BMTs over a set of files F . Let $\cap$ be the (standard) tree join defined as follows:

$$\alpha \cap \beta = \text{node } \alpha \text{ } (\alpha.\text{val} + \beta.\text{val}) \text{ } \beta$$

We define $\cap_{\leq}$ as the minimum cost join on members of M_F as follows:

$$\alpha \cap_{\leq} \beta = \begin{cases} \alpha & \text{Is-empty}(\beta) \\ \alpha \cap \beta & \text{Is-leaf}(\alpha) \wedge \text{Is-leaf}(\beta) \\ \delta \cap_{\leq} (\sigma \cap_{\leq} \beta) & \alpha = \delta \cap_{\leq} \sigma \\ \alpha \cap \beta & \beta = \lambda \cap \rho \wedge \lambda.\text{val} \leq \alpha.\text{val} \wedge \rho.\text{val} \leq \alpha.\text{val} \\ (\alpha \cap_{\leq} \lambda) \cap \rho & \beta = \lambda \cap \rho \wedge \alpha.\text{val} \leq \rho.\text{val} \wedge \lambda.\text{val} \leq \rho.\text{val} \\ \lambda \cap (\alpha \cap_{\leq} \rho) & \beta = \lambda \cap \rho \wedge \alpha.\text{val} \leq \lambda.\text{val} \wedge \rho.\text{val} \leq \lambda.\text{val} \end{cases}$$

The identity of this operator is the null BMT denoted by *empty*.

Since $\cap_{\leq}$ operates on elements of M_F , its arguments are minimal cost BMTs. In the first two cases it is obvious that what results is also minimal cost. If neither of these cases hold then our result will be one of $\alpha \cap \beta$, $(\alpha \cap_{\leq} \lambda) \cap \rho$, $\lambda \cap (\alpha \cap_{\leq} \rho)$. There are no other ways in which we can combine the trees. Let us examine each of these in turn.

$$\begin{aligned} \text{Cost}(\alpha \cap \beta) &= \alpha.\text{val} + \lambda.\text{val} + \rho.\text{val} + \text{Cost}(\alpha) + \lambda.\text{val} + \rho.\text{val} + \text{Cost}(\lambda) + \text{Cost}(\rho) \\ \text{Cost}((\alpha \cap_{\leq} \lambda) \cap \rho) &= \alpha.\text{val} + \lambda.\text{val} + \rho.\text{val} + \alpha.\text{val} + \lambda.\text{val} + \text{Cost}(\alpha) + \text{Cost}(\lambda) + \text{Cost}(\rho) \\ \text{Cost}(\lambda \cap (\alpha \cap_{\leq} \rho)) &= \alpha.\text{val} + \lambda.\text{val} + \rho.\text{val} + \text{Cost}(\lambda) + \alpha.\text{val} + \rho.\text{val} + \text{Cost}(\alpha) + \text{Cost}(\rho) \end{aligned}$$

From this it should be clear that when $\lambda.\text{val} + \rho.\text{val} \leq \alpha.\text{val} + \lambda.\text{val}$ and $\lambda.\text{val} + \rho.\text{val} \leq \alpha.\text{val} + \rho.\text{val}$ then the join $\alpha \cap \beta$ is the least expensive option. This condition can be simplified to $\lambda.\text{val} \leq \alpha.\text{val} \wedge \rho.\text{val} \leq \alpha.\text{val}$. Similar reasoning motivates the guards for the other 2 cases. Thus the set M_F is closed under $\cap_{\leq}$.

The reader should recognise that this captures the strategy employed in the ad-hoc traditional solution, namely, merge the smallest files first and leave the largest ones until as late as possible.

The $\cap_{\leq}$ operator is partially defined in terms of a standard tree join $\cap$. $\cap$ is not associative, yet $\cap_{\leq}$ is.

We have characterised the monoid of minimal cost BMTs ($M_F, \cap_{\leq}, \text{empty}$). In the next section we use this to construct a solution to the file merge problem.

2.6 Constructing the optimal merge strategy

Given a set of files $F_0, \dots, F_{n-1}$ to be merged, it is trivial to inject them into the corresponding sequence $\langle F_0, \dots, F_{n-1} \rangle$. In addition, we make use of the inject function *Entree*: $F \rightarrow M_F$ defined as follows:

$$\text{Entree}(f) = \text{node empty} \# f \text{ empty}$$

where $\#f$ denotes the length of file f .

Given the free monoid over the set of ordered files (*Ordered-Files*^{*}, $\wedge$, $\langle \rangle$) there exists a monoid homomorphism to the monoid of minimal cost BMTs ($M_F, \cap_{\leq}, \text{empty}$):

$$\text{Gen-Strategy}: (\text{Ordered-Files}^*, \wedge, \langle \rangle) \rightarrow (M_F, \cap_{\leq}, \text{empty})$$

which may be expressed as a recursive function that computes the optimal merge strategy for a sequence of files:

$$\begin{aligned} \text{Gen-Strategy} \langle \rangle &= \text{empty} \\ \text{Gen-Strategy} \langle a \rangle \wedge B &= \text{Entree}(a) \cap_{\leq} \text{Gen-Strategy}(B) \end{aligned}$$

For N files, the worst-case complexity of this approach is $O(N \times \xi)$ where ξ denotes the worst-case complexity of the $\cap_{\leq}$ operator.

3 A second example: minimum coin configurations

Suppose, given an unlimited supply of coins of particular denominations and a certain amount of money, we are required to represent that amount using the smallest number of coins possible.

For example, assume that the denominations available are 1, 2, 5 and 10, and the amount to be represented is X . This problem will be familiar to anyone who has worked on a computerised payroll system of the '60s or '70s. The solution usually proposed is the following: firstly, maximise the number of denomination 10 coins: $X \text{ div } 10$, leaving as remainder $X \text{ mod } 10$. Follow a similar strategy for the denominations 5 and 2 and the remainder at that stage is represented by denomination 1 coins. This succeeds because of a number of properties of the denominations. For the moment, assume that we aren't aware of these properties.

For a given amount X there may be a number of possible representations using the coins. Let us represent such a configuration as a 4-tuple whose elements from left to right indicate the number of coins of progressively larger denomination. For example, the amount 12 is be represented by any of the following 4-tuples:

$$\begin{aligned} (12, 0, 0, 0) \\ (0, 6, 0, 0) \\ (2, 0, 2, 0) \\ (2, 0, 0, 1) \\ \text{etc.} \end{aligned}$$

These are all valid representations but we can order them by the number of coins used, giving rise to a

greatest lower bound which represents the optimal configuration.

Given two optimal configurations, corresponding to amounts i and j , we seek a composition operator $\oplus$ that yields the optimal configuration for the sum $i+j$. The identity element of $\oplus$ should be the 4-tuple $(0, 0, 0, 0)$.

Suppose that A and B are optimal coin configurations. Our definition restricts A to be $(1, 0, 0, 0)$. This is the optimal configuration representing the amount 1, so whatever amount we wish A to represent can be achieved by successive compositions of $(1, 0, 0, 0)$.

The following Markov algorithm (see [10] for an overview) describes the behaviour of $\oplus$ where the left-hand side of the productions describe the possible patterns of B :

$$\begin{aligned} 0, 2, 1, i &\rightarrow . 0, 0, 0, i+1 \\ 0, 2, i, j &\rightarrow . 0, 0, i+1, j \\ 1, i, j, k &\rightarrow . 0, i+1, j, k \\ 0, i, j, k &\rightarrow . 1, i, j, k \end{aligned}$$

We allow i, j and k to represent arbitrary natural numbers.

To arrive at this algorithm we examine each of the ranks corresponding to denominations 1, 2, 5, 10 and characterise the circumstances under which that rank is increased by 1 if $(1, 0, 0, 0)$ is added to the amount represented. We only consider those circumstances which are themselves optimal. We then arrange the productions in such a way that those which cause the least increase (or the most decrease) in the number of coins trigger first. Within that we order the transformations so that those which increase the number of coins of a higher denomination trigger before those which increase the numbers of a lower denomination.

For a given amount of money there exists but one optimal configuration and, since this join maintains optimality, this suggests that the join is associative.

Let C_{opt} denote the set of optimal coin configurations. It is closed under $\oplus$ which is an associative operator with identity $(0, 0, 0, 0)$. This is the monoid of optimal coin configurations for coins of these denominations.

There exists a homomorphism *Optimise*: $(\mathbf{N}, +, 0) \rightarrow (C_{opt}, \oplus, (0, 0, 0, 0))$. Using it we can construct the following recursive function which takes a given amount and generates the optimal coin configuration:

$$\begin{aligned} \text{Optimise}(0) &= (0, 0, 0, 0) \\ \text{Optimise}(n+1) &= (1, 0, 0, 0) \oplus \text{Optimise}(n) \end{aligned}$$

which has worst-case complexity $O(N \times \xi)$ where ξ denotes the worst-case complexity of $\oplus$.

The reader may ask why we have gone to this trouble to construct a solution given that we could use the simpler approach using **div** and **mod**. We agree that this would have been easier but such an approach does not generalise to all other denominations. In the next section we consider a variation on the problem where this is the case.

3.1 Minimum configurations using non-standard denominations

Let us add a further coin denomination with the value 8. Using the denominations 1, 2, 5, 8, 10 we shall attempt to show how to produce minimum coin configurations.

We represent a configuration as a 5-tuple. Notice that a difficulty arises if we adopt the simple approach described in the last section. Suppose we are given the amount 13. An approach which always seeks to use the largest denomination wherever possible would give us $(1, 1, 0, 0, 1)$ but we can achieve our goal using one less coin: $(0, 0, 1, 1, 0)$.

$\oplus$ has the identity $(0, 0, 0, 0, 0)$ and is defined as follows (again assuming that A is $(1, 0, 0, 0, 0)$ and the left-hand side of the productions represent configurations of B):

$$\begin{aligned} 1, 0, 0, i+1, j &\rightarrow . 0, 0, 0, i, j+1 \\ 0, 1, 0, i, j+1 &\rightarrow . 0, 0, 1, i+1, j \\ 0, 1, 1, i, j &\rightarrow . 0, 0, 0, i+1, j \\ 0, 2, 0 &\rightarrow . 0, 0, 1 \\ 1, i &\rightarrow . 0, i+1 \\ 0 &\rightarrow . 1 \end{aligned}$$

To arrive at these productions we consider each rank and determine the possible configurations under which adding 1 to the configuration results in the value at that rank being increased. Once again we only consider configurations on the left hand side which are optimal and only chose for the right hand side those which maintain optimality. As before we rank the productions so as to favour increases to coins of higher denomination.

Again, we claim that the operator $\oplus$ is associative because it always produces the optimal configuration for a particular amount and thus the manner in which we compose smaller amounts to arrive at this configuration cannot change the optimality. As the set of optimal configurations is closed under $\oplus$ we have a monoid.

Again, to obtain an optimal coin configuration for a given amount we can construct a recursive function based on the homomorphism between $(\mathbf{N}, + 0)$ and this monoid.

4 Conclusions

We have examined some familiar optimisation problems which may be considered trivial. However, the conceptual model we have used in their solution differs from the usual approaches. We believe that there may be merit in exploring the types of monoid described above.

When we can characterise the solution to a problem as an instance of a structure of a particular type, it seems natural to examine that structure and ask whether there is a composition operator for it. If such an operator exists, we ask whether the set of instances of this structure is closed under it, whether it has an identity, whether each instance of the structure has an inverse, and so on.

It seems that in traditional approaches we begin with a problem instance, say P_k , of k and seek to map this to its solution, S_k . However, this mapping may be difficult to find. Moreover, in looking for it we may not spend much time studying either the problem or solution domains.

Our approach differs in that we ask whether the problem P_k may be viewed as the composition of problems P_i and P_j , using a suitable composition operator. These sub-problems yield solutions S_i and S_j . Again we ask whether there is a suitable join operator on the solution domain such that we can compose S_i and S_j to give us S_k .

We believe that trying to find these composition operators is a worthwhile undertaking. At worst, it will contribute to a deeper understanding of the problem and the nature of its solution; better than this, it may contribute to the discovery of elegant solutions.

Furthermore, if both the problem and solution domains are monoids, then the homomorphism between them (if such exists) can be expressed in a nice recursive form where a solution of size k is expressed in terms of successively composing solutions of size 1. Whilst the homomorphism may be quite complicated, in many cases the solution for problems of size 1 turns out to be easy to find. This recursive solution has worst-case complexity $O(k \times \xi)$ where ξ is the complexity of the composition operator on the solution domain.

It is our intention to explore a range of such optimisation problems to determine the extent to which adopting this conceptual model helps shed light on them and contributes to their solution. The most common traditional approaches used in solving optimisation problems are dynamic programming and greedy algorithms. Both of these approaches should be familiar to the reader. We chose not to discuss these methods here. Our aim in this paper has been simply to describe how a different conceptual model *might* be used in solving optimisation problems and we are not yet in a position to claim that this approach is better or worse than the traditional approaches. Such a comparison will be performed in a future, more rigorous, paper.

5 References

- [1] Baase S. Computer Algorithms: Introduction to Design & Analysis, Addison-Wesley, 1988.
- [2] Cohen E. Programming in the 1990s, Springer-Verlag, 1990.
- [3] Dijkstra E.W., Feijen, W.H.J. A Method of Programming, Addison-Wesley, 1988.
- [4] Gries D. The Science of Programming, Springer-Verlag, 1981.
- [5] Kaldewaij A. Programming: The Derivation of Algorithms, Prentice-Hall, 1990.
- [6] Mac an Airchinnigh M. Conceptual Models and Computing, PhD thesis, Trinity College, University of Dublin, 1990.
- [7] Mac an Airchinnigh M. Tutorial Lecture Notes on the Irish School of the VDM. In Prehn S., Toetenel W. J. (eds) VDM '91—Formal Software Development Methods, Springer-Verlag, 1991, pp 141-237, LNCS 552.
- [8] Mac an Airchinnigh M. Formal Methods and Testing, Technical Report, Department of Computer Science, Trinity College, University of Dublin, May 1993.
- [9] McLoughlin H., Hely K. Teaching Formal Programming to First Year Computer Science Students, Proceedings of the Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education, ACM Press, 1996.

- [10] Taylor R.G. Models of Computation and Formal Languages, Oxford University Press, 1998.