

Constructing Datatype-Generic Fully Polynomial-Time Approximation Schemes Using Generalised Thinning

Shin-Cheng Mu Yu-Han Lyu

Institute of Information Science
Academia Sinica, Taiwan
scm@iis.sinica.edu.tw, yuhanlyu@gmail.com

Akimasa Morihata

RIEC, Tohoku University, Japan
moriyata@riec.tohoku.ac.jp

Abstract

The *fully polynomial-time approximation scheme* (FPTAS) is a class of approximation algorithms that is able to deliver an approximate solution within any chosen ratio in polynomial time. By generalising Bird and de Moor's *Thinning Theorem* to a property between three orderings, we come up with a datatype-generic strategy for constructing fold-based FPTASs. Greedy, thinning, and approximation algorithms can thus be seen as a series of generalisations. Components needed in constructing an FPTAS are often natural extensions of those in the thinning algorithm. Design of complex FPTASs is thus made easier, and some of the resulting algorithms turn out to be simpler than those in previous works.

Categories and Subject Descriptors F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

General Terms Algorithms, Theory

Keywords Approximation Algorithms, Program Derivation

1. Introduction

Program derivation aims to develop and advocate a methodology with which programs can be formally constructed from their specifications. A particularly fruitful series of results was Bird and de Moor's relational, datatype generic approach for solving optimisation problems [4]. An optimisation problem is specified as a composition $max \trianglelefteq \circ solutions$ where $solutions$ generates all solution candidates, and $max \trianglelefteq$ picks an optimal solution under a total preorder \trianglelefteq (the notations will be explained in Section 2). It was shown that if $solutions$ can be written as a relational fold whose step relation is monotonic with respect to \trianglelefteq , there is a greedy algorithm solving the problem. The monotonicity condition might not always hold, but for many problems the function $solutions$ provides some hint to find a non-total sub-ordering of \trianglelefteq for which the monotonicity condition does hold. In this case we have a so-called "thinning" algorithm (the "fold" counterpart of dynamic programming). The theorems are datatype-generic and applicable to problems defined on a wide collection of inductively defined datatypes. Their work not only hinted at a methodology to construct efficient

algorithms to optimisation problems whose proofs are developed together with the programs, but also made the relationship between different classes of algorithms formal and clear.

One of the missing pieces of the jigsaw puzzle is a formal treatment of approximation algorithms. One may be interested in approximation algorithms out of practical concern: by sacrificing some precision, one may find provably near-optimal solutions to NP-hard problems in polynomial time. Besides the pragmatic, approximation algorithms are of interest to the theoreticians as well. It is believed that some NP-hard problems can be approximated within any factor, some can be approximated by a constant, yet some are impossible to approximate within any constant at all, unless $P = NP$. Such classification could therefore reveal important clues to tackle some long standing theoretical issues.

The *fully polynomial-time approximation scheme* (FPTAS) [9, 11, 12, 16] is a class of approximation algorithms that delivers an approximate solution no worse than the optimal solution within any chosen ratio in time polynomial in the size of the input and the inverse of the ratio (precise definition to be given in Section 4). Structures of some FPTAS algorithms are surprisingly similar to the thinning strategy of Bird and de Moor specialised to lists, which lead us to believe that the formal development can be extended to approximation algorithms, and that FPTAS can be naturally extended to more datatypes. As it turns out, if it is known how to solve an optimisation problem using a thinning strategy, an FPTAS can often be constructed as a fold by extending the ordering used for thinning. Our contributions include:

- We propose a generalisation of the Thinning Theorem, from one preorder to three relations, to prove correctness of fold-based FPTAS. We have thus filled in another piece of the algorithm genealogy, seeing the route from greedy, to thinning, to approximation algorithms as a series of generalisations.
- We present an algebraic formalism for a certain class of approximation algorithms whose correctness is not trivial to prove otherwise. Our formalism is, to the best of our knowledge, the first datatype generic formulation of FPTAS. Yet it is simpler in structure than some existing FPTAS formulations.
- The formalism suggests a methodology for developing FPTAS, often considered a non-trivial task, by extending the ordering used in the corresponding thinning algorithm in a natural and sometimes the only possibly way.
- We demonstrate derivations of several approximation algorithms, some of which surprisingly turn out to be simpler than those in previous works.

In Section 2 we give a quick introduction to the relational approach to optimisation problems and the thinning strategy, using 0-1 knapsack as an example. Section 3 presents our generalisation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WGP'10, September 26, 2010, Baltimore, Maryland, USA.
Copyright © 2010 ACM 978-1-4503-0251-7/10/09...\$10.00

of the Thinning Theorem, which we use to model FPTAS in Section 4. We apply our theorems to three examples in Section 5: the approximate version of 0-1 knapsack, a task scheduling problem, and an NP-hard optimisation problem on trees. Code accompanying this paper is available online [15].

2. Review: Relational Approach to Optimisation Problems

In this section we give a minimal review of the basic concepts we need. The reader is referred to Bird, Gibbons and Mu [5] for a short tutorial on relational program calculation for optimisation problems, and to Bird and de Moor [4] for a thorough introduction from a categorical perspective.

2.1 Sets, Functions, and Relations

A relation R from set A to set B , written $R :: B \leftarrow A$, is a subset of the set $\{(b, a) \mid b \in B \wedge a \in A\}$.¹ The direction of the arrow is chosen to be consistent with relational composition. When $(b, a) \in R$, we say R maps a to b and sometimes also write $b \leftarrow R a$, intuitively meaning that b is a result that R maps a to. In this paper we will often define a relation using the notation:

$$b \leftarrow R a \equiv \dots$$

(where \equiv denotes “if and only if”) meaning that the relation R consists of exactly those (b, a) for which the conditions on the right-hand side holds. A function f is a special case of a relation such that $b \leftarrow f a$ and $b' \leftarrow f a$ implies $b = b'$. We use the usual forward arrow $f :: A \rightarrow B$ when we emphasise that f is a function.

Since relations are sets, set operations such as union, intersection, and subset ordering apply to relations as well. Given relations $R :: C \leftarrow B$ and $S :: B \leftarrow A$, their composition $R \circ S :: C \leftarrow A$ is defined by:

$$c \leftarrow (R \circ S) a \equiv (\exists b : c \leftarrow R b \wedge b \leftarrow S a).$$

Composition is monotonic with respect to set inclusion. The identity relation $id a = a$ is the unit of composition. Given a predicate p , the relation $p?$ is a sub-relation of id defined by:

$$a \leftarrow p? a \equiv p a.$$

It is often used to filter results satisfying certain conditions.

The *power transpose* operator Λ converts a relation to a set-valued total function:

$$(\Lambda R) a = \{b \mid (b, a) \in R\}.$$

Also note that \in is a relation, written infix, mapping a set to one of its arbitrary elements. For a definition, \in is the set of all (a, x) where a is a member of x .

The disjoint sum of two sets A and B is defined by $A + B = \{inl a \mid a \in A\} \cup \{inr b \mid b \in B\}$. Given $R :: C \leftarrow A$ and $S :: C \leftarrow B$, their “join” $[R, S] :: C \leftarrow (A + B)$ is defined by:

$$\begin{aligned} c \leftarrow [R, S] (inl a) &\equiv c \leftarrow R a \\ c \leftarrow [R, S] (inr b) &\equiv c \leftarrow S b, \end{aligned}$$

while the sum functor is defined by $R + S = [inl \circ R, inr \circ S]$. The Cartesian product of two sets $A \times B$ is defined by $\{(a, b) \mid a \in A \wedge b \in B\}$. Let $fst(a, b) = a$ and $snd(a, b) = b$. The “split” operator and the product functor are defined respectively by:

$$\begin{aligned} (b, c) \leftarrow \langle R, S \rangle a &\equiv b \leftarrow R a \wedge c \leftarrow S a, \\ (R \times S) &= \langle R \circ fst, S \circ snd \rangle. \end{aligned}$$

Functional programmers may be more familiar with the special case when both arguments are functions: $\langle f, g \rangle a = (f a, g a)$, and $(f \times g)(a, b) = (f a, g b)$.

The relation $distr$ maps (x, ys) to all (x, y) if $y \in ys$:

$$distr = (id \times \in).$$

Therefore, we have $(\Lambda distr)(x, ys) = \{(x, y) \mid y \in ys\}$.

2.2 Relational Folds on Regular Datatypes

Researchers working on datatype-generic programming are familiar with the initial algebra construction of regular datatypes. An *F-algebra* (A, f) for a functor F consists of a type A (called the “carrier”) and a function $f :: F A \rightarrow A$. An *F-algebra* (T, in_T) is *initial* if for all *F-algebra* (A, f) there is a unique homomorphism $fold_F f :: T \rightarrow A$ such that

$$fold_F f \circ in_T = f \circ F(fold_F f).$$

Such an initial algebra is known to exist for a class of functors that are called *regular* — those that can be defined in terms of 1 (the unit type), sum, product, and type constructors.

For a typical example, the type of inductively defined lists $[A]$ can be seen as the carrier of the initial algebra of functor $FX = 1 + A \times X$. For every function $f :: (1 + A \times B) \rightarrow B$ there exists a unique homomorphism $fold_F f :: [A] \rightarrow B$. A function f of this type can always be decomposed into $[const\ base, step]$ with $step :: (A \times B) \rightarrow B$ and $base :: B$, where $const$ is the constant function. We have thus recovered the usual *foldr* on lists by letting $foldr\ step\ e = fold_F [const\ e, step]$:

$$\begin{aligned} foldr :: ((A \times B) \rightarrow B) \rightarrow B \rightarrow [A] \rightarrow B \\ foldr\ f\ e [] &= e \\ foldr\ f\ e (a : x) &= f(a, foldr\ f\ e\ x). \end{aligned}$$

Another datatype we will see in the example is that of rose trees:

$$\mathbf{data}\ Tree\ A = N\ A\ [Tree\ A],$$

which can be obtained by taking $FX = A \times [X]$, and give rise to the following fold:

$$\begin{aligned} foldT :: ((A \times [B]) \rightarrow B) \rightarrow Tree\ A \rightarrow B \\ foldT\ f\ (N\ a\ ts) &= f\ a\ (map\ (foldT\ f)\ ts). \end{aligned}$$

The family of $fold_F$ enjoys a useful property called *fold fusion law*:

$$g \circ f = f' \circ Fg \Rightarrow g \circ fold_F f = fold_F f'.$$

With a little twist², the concept can be generalised to relations. Given a relation $R :: B \leftarrow F B$, the relational $fold_F :: B \leftarrow T B$ is defined in terms of functional fold:

$$fold_F R = \in \circ fold_F \Lambda(R \circ F \in).$$

Note that the right-hand side refers to the functional $fold_F$, thus the definition above is not recursive. The fold fusion rule for relational folds generalises to inclusion:

$$R \circ S \supseteq T \circ FR \Rightarrow R \circ fold_F S \supseteq fold_F T.$$

Let us see some instantiation to specific datatypes. Since all relations $R :: B \leftarrow (1 + A \times B)$ can be factored to $[\in \circ const\ s, S]$ with $S :: B \leftarrow (A \times B)$ and $s :: \{B\}$, the relational fold for lists of type $[A]$ is given by:

$$foldr\ S\ s = \in \circ foldr\ \Lambda(S \circ distr)\ s.$$

The functional fold on the right-hand side returns a set $\{B\}$ of all the results, before \in picks an arbitrary element. For the base case, the functional fold returns s ; for input $a : x$, the relation $distr$ pairs

¹ We use a Haskell-like notation denoting “has type” by $::$.

² Namely, using the Cartesian product defined in Section 2.1 in place of categorical products. See Bird and de Moor [4].

the current element a with each result computed from x and passes the pair to S .

For an example, let y be a *sublist* of x if y contains some, all, or none of the elements of x , preserving their order. The relation *sublist*, mapping the input to one of its sublists, can be defined as a relational fold:

$$\text{sublist} = \text{foldr} (\text{cons} \cup \text{snd}) [], \quad (1)$$

where $\text{cons} (a, x) = a : x$ keeps the current element, while snd drops it.

2.3 Maximum, Monotonicity, and Thinning

A relation of type $A \leftarrow A$ for some A is called a *preorder* if it is reflexive and transitive. We use the infix notation $a \trianglelefteq b$ to denote $(a, b) \in \trianglelefteq$. A preorder \trianglelefteq is said to be *connected* if for all $a, b \in A$, either $a \trianglelefteq b$ or $b \trianglelefteq a$. The relation $\text{max} \trianglelefteq$ picks a maximum element from the input set:

$$a \rightsquigarrow (\text{max} \trianglelefteq) xs \equiv a \in xs \wedge (\forall b \in xs : b \trianglelefteq a).$$

The definition itself assumes nothing of \trianglelefteq . However, some theorems we will introduce later demand \trianglelefteq to be a preorder.

Optimisation Problem We specify an optimisation problem by

$$\text{max} \trianglelefteq \circ \Lambda \text{solutions}.$$

The relation $\text{solutions} :: B \leftarrow A$ maps an input of type A to an arbitrary solution of type B . We can integrate the test of feasibility into solutions — the feasible inputs are those in its domain. The relation $\text{max} \trianglelefteq$ picks a maximum, under ordering \trianglelefteq , among the set of all the solutions. Given a function $f :: B \rightarrow \mathbb{R}$, define

$$x \leq_f y \equiv f x \leq f y.$$

Many optimisation problems are about picking a maximal solution under relation \leq_f for some f . The function f is often called the *objective function*.

For this paper, if a relation is written infix using an “ordering-like” symbol, say \trianglelefteq , we denote its converse by \trianglerighteq (that is, $x \trianglerighteq y \equiv y \trianglelefteq x$). A minimisation problem can be specified using the relation min , defined by $\text{min} \trianglelefteq = \text{max} \trianglerighteq$. Since we will be using two maximisation problems as examples, most of our results will be stated in terms of max but apply to min as well.

Fold-based Problems Bird and de Moor [4] considered optimisation problems that can be expressed in the following form:

$$\text{max} \trianglelefteq \circ \Lambda(\text{fold}_F R). \quad (2)$$

That is, those problems whose set of possible solutions can be generated by a relational fold. For an example, consider the special case of lists, for which (2) expands to

$$\text{max} \trianglelefteq \circ \text{foldr} \Lambda(S \circ \text{distr}) s,$$

meaning that s is the set returned for the empty list, while new potential solutions are generated from old ones by $\Lambda(S \circ \text{distr})$ in the non-empty case. To obtain an efficient algorithm, one would like to maintain a smaller set of solutions. So let us find out conditions under which some solutions bound to be sub-optimal can be dropped.

Monotonicity A relation $R :: FB \rightarrow B$ is said to be *monotonic on* \trianglelefteq if

$$R \circ F \trianglelefteq \subseteq \trianglelefteq \circ R.$$

In words, if $x F \trianglelefteq x'$ and $y \rightsquigarrow R x$, there exists some y' such that $y' \rightsquigarrow R x'$ and $y \trianglelefteq y'$. It requires that R maps larger inputs to larger outputs. For the special case of lists, where R can be factored into $[\epsilon \circ \text{const } s, S]$ where $S :: (A \times B) \rightarrow B$, the monotonicity

condition instantiates to that for all $a :: A$ and $b_1, b_2, c_1 :: B$:

$$c_1 \rightsquigarrow S(a, b_1) \wedge b_1 \trianglelefteq b_2 \Rightarrow (\exists c_2 : c_1 \trianglelefteq c_2 \wedge c_2 \rightsquigarrow S(a, b_2)). \quad (3)$$

For some intuition, assume that we want a solution that is generated from iterated applications of S and is maximal with respect to \trianglelefteq , and consider two potential solutions b_1 and b_2 such that $b_1 \trianglelefteq b_2$. That S is monotonic on \trianglelefteq means that for every c_1 we can generate from $S(a, b_1)$, there must be some c_2 resulting from $S(a, b_2)$ that is no smaller than c_1 . There is thus no point keeping b_1 .

Given an optimisation problem specified in the form of (2), if R is monotonic on \trianglelefteq , in each step of $\text{fold}_F R$ we only need to keep the best solution with respect to \trianglelefteq , which leads to a greedy algorithm. Notice, however, that \trianglelefteq has to be connected for *the* best solution to be defined.

Thinning It is often the case that R is only monotonic on a non-connected subset \preceq of \trianglelefteq . An algorithm thus needs to maintain at least those solutions that are *maximal* under \preceq . Rather than giving a specification that strictly keeps only those maximal elements, it turns out to be more flexible to specify a relation that may or may not drop some useless solutions. Given a relation \preceq , let $\text{thin} \preceq$ be a relation between sets of solutions so that for all xs and ys :

$$ys \rightsquigarrow \text{thin} \preceq xs \equiv ys \subseteq xs \wedge (\forall b \in xs : (\exists b' \in ys : b \preceq b')).$$

One way to understand it is that $\text{thin} \preceq$ maps a set to one of its “safe” subsets — ys is a safe subset of xs if for every b in xs , there is some b' remaining in ys that is no smaller than b under \preceq .

Given \trianglelefteq and a relation \preceq that is a sub-relation of \trianglelefteq (that is, $x \preceq y \Rightarrow x \trianglelefteq y$), we have:

$$\text{max} \trianglelefteq = \text{max} \trianglelefteq \circ \text{thin} \preceq. \quad (4)$$

This allows us to refine the specification (2) to $\text{max} \trianglelefteq \circ \text{thin} \preceq \circ \Lambda(\text{fold}_F R)$. We may then apply the *Thinning Theorem* given by Bird and de Moor [3]:

Theorem 1. *If R is monotonic on a transitive relation \preceq , then:*

$$\text{thin} \preceq \circ \Lambda(\text{fold}_F R) \supseteq \text{fold}_F (\text{thin} \preceq \circ \Lambda(R \circ F \epsilon)).$$

The left-hand side of \supseteq generates the set of all solutions by a fold as before. The set is then related to all safe subsets by $\text{thin} \preceq$. The inclusion means if we move the thinning *into* the fold, thereby reducing the size of sets we maintain at each stage, we still get a safe set of solutions.

Presented in a datatype-generic style, the Thinning Theorem applies to a wide range of datatypes that can be defined by initial algebras. With the theorem, the algorithm designer is left with the job of coming up with a (usually problem-specific) ordering \preceq and a proof of monotonicity, while the theorem takes care of transforming the specification to the right form.

It is not specified how thoroughly the set of solutions should be thinned — the right-hand side of Theorem 1 can be further refined to an algorithm by replacing $\text{thin} \preceq$ with any function that is its sub-relation and, by transitivity of \supseteq , we know that the algorithm still delivers a safe set. In this sense the theorem covers a wide spectrum of algorithms. On one extreme, *id* is a sub-relation of $\text{thin} \preceq$, leading to a brute force algorithm that generates all solutions. On the other extreme, we can refine $\text{thin} \preceq$ to a function that maintains exactly the maximal elements under \preceq , but it may be costly to thoroughly eliminate all sub-optimal elements. In practice, we usually try to come up with a representation of sets that allows us to perform thinning in time proportional to the size of the set.

2.4 Example: 0-1 Knapsack

The 0-1 knapsack problem is often used as an example of dynamic programming. A typical solution to the 0-1 knapsack prob-

lem builds a table of solutions indexed by their values. In this section we will see how it can be solved by a thinning strategy. Given a set of *Items* with two objective functions: $value, weight :: Item \rightarrow \mathbb{R}^+$, the goal is to pick a subset of items (a “packing”) whose total weight is under some weight limit W and whose total value is as large as possible. For simplicity we overload $value$ and $weight$ to lists of items when there is no confusion:

$$\begin{aligned} value, weight &:: [Item] \rightarrow \mathbb{R} \\ value &= sum \circ map \ value, \\ weight &= sum \circ map \ weight, \end{aligned}$$

and abbreviate \leq_{value} and \leq_{weight} respectively to \leq_v and \leq_w . The 0-1 knapsack problem can be specified by:

$$\begin{aligned} knapsack &:: [Item] \rightarrow [Item] \\ knapsack &= max \leq_v \circ \Lambda(\text{safe?} \circ \text{sublist}), \end{aligned}$$

where *sublist* is defined in (1), and the predicate *safe* is defined by $(\leq W) \circ weight$. By relational fold-fusion, *safe?* and *sublist* can be fused into one:

$$knapsack = max \leq_v \circ \Lambda(\text{foldr} (scons \cup \text{snd}) \{\{\}\}),$$

where $(a : x) \leftarrow scons(a, x) \equiv weight(a : x) \leq W$. The specification is thus in the form of (2).

Thinning The monotonicity condition (3), with $S := scons \cup \text{snd}$, instantiates to:

$$\begin{aligned} ((y_1 = a : x_1 \wedge \text{safe } y_1) \vee y_1 = x_1) \wedge x_1 \leq x_2 \Rightarrow \\ (\exists y_2 : ((y_2 = a : x_2 \wedge \text{safe } y_2) \vee y_2 = x_2) \wedge y_1 \leq y_2). \end{aligned}$$

To see that it does not hold when $\leq := \leq_v$, consider two packings x_1 and x_2 with $x_1 \leq_v x_2$, while x_2 is heavier than x_1 . On the left-hand side of \Rightarrow we pick $y_1 = a : x_1$, that is, adding some a to x_1 through *scons*. On the right-hand side, however, it could be the case that a cannot be added to x_2 because *safe* ($a : x_2$) does not hold, and $y_1 \geq_v x_2$.

One may check, however, that $scons \cup \text{snd}$ is monotonic on \leq defined below:

$$x_1 \leq x_2 \equiv x_1 \leq_v x_2 \wedge x_1 \geq_w x_2.$$

By adding an extra constraint $x_1 \geq_w x_2$ we ensure that no matter how we extend x_1 , we can always extend x_2 the same way and yield a packing that is at least as good. We thus have:

$$\begin{aligned} knapsack &\supseteq \{ \text{since } \leq \leq_v, \text{ by (4)} \} \\ &max \leq_v \circ \text{thin } \leq \circ \Lambda(\text{foldr} (scons \cup \text{snd}) \{\{\}\}) \\ &\supseteq \{ scons \cup \text{snd} \text{ monotonic on } \leq, \text{ Theorem 1} \} \\ &max \leq_v \circ \text{foldr} (\text{thin } \leq \circ \Lambda((scons \cup \text{snd}) \circ \text{distr})) \{\{\}\}. \end{aligned}$$

Refining to Function The calculation above hints at that we may develop an algorithm that maintains a set of possible packings and, at each step, removes some packings that are dominated by others under the ordering \leq . For efficiency, however, it is crucial to come up with a way to represent the set so that thinning can be efficient.

One possible strategy is to represent the set as a list of packings sorted in descending order by their values. The function *merge* in Figure 1 takes two lists descendingly sorted by \leq_f for some objective function f and merges them, in time proportional to the lengths of the lists, into one sorted list. The function *bump* assumes that the input is sorted by their images on function f . For each f -value, it keeps at most one element having minimal value on g . The tests surrounded by boxes are guaranteed to be true in their contexts and are written there for clarity.

$$\begin{aligned} merge \ f \ [] \ ys &= ys \\ merge \ f \ xs \ [] &= xs \\ merge \ f \ (x : xs) \ (y : ys) & \\ & \quad | \ x \geq_f \ y = x : merge \ f \ xs \ (y : ys) \\ & \quad | \ otherwise = y : merge \ f \ (x : xs) \ ys \end{aligned}$$

$$\begin{aligned} bump \ f \ (\leq_g) \ [] &= [] \\ bump \ f \ (\leq_g) \ [x] &= [x] \\ bump \ f \ (\leq_g) \ (x : y : xs) & \\ & \quad | \ x \leq_g \ y \ \wedge \ x \geq_f \ y = bump \ f \ (\leq_g) \ (x : xs) \\ & \quad | \ \boxed{x \geq_g \ y \ \wedge \ x =_f \ y} = bump \ f \ (\leq_g) \ (y : xs) \\ & \quad | \ \boxed{x \geq_g \ y \ \wedge \ x >_f \ y} = x : bump \ f \ (\leq_g) \ (y : xs) \end{aligned}$$

Figure 1. List merging and bumping.

With *merge* and *bump* defined, a thinning algorithm for the 0-1 knapsack problem can be implemented by:

$$\begin{aligned} knapsack &= head \circ \text{foldr} \ step \ \{\{\}\} \\ step \ a \ xs &= bump \ value \ (\leq_w) \\ & \quad (merge \ value \ xs \ [a : x \mid x \leftarrow xs, weight(a : x) \leq W]). \end{aligned}$$

The first occurrence of xs in the last line corresponds to the result generated by *snd*, while the list comprehension corresponds to that by *scons*. Since the *foldr* produces a list sorted by values, $max \leq_v$ in the specification is refined to *head*.

Efficiency The program maintains a list that plays the same role as the typical value-indexed table, with some slight advantages. De Moor [7] claimed that, in a lazy setting, the program outperforms the traditional table-based algorithm.

We give an analysis when all weights and values are integral. Note that after *bump*, the solutions are sorted by strictly increasing value and, lexicographically, strictly increasing weight. Moreover, the list of solutions does not contain those with weight heavier than W . In the worst-case scenario, the size of the list of solutions is still bounded by the minimum of W and $V = sum(map \ value \ x)$. Both *merge* and *bump* takes time proportional to the length of the lists, thus the algorithm takes $O(n \cdot (W \downarrow V))$ time, where (\cdot) denotes number multiplication and (\downarrow) denotes minimum.³ The algorithm is thus *pseudo-polynomial* — it runs in time polynomial in the numerical value of the input, and therefore exponential in the length of the input.

3. Generalised Thinning

Consider Theorem 1 again. One way to interpret the conclusion

$$\text{thin } \leq \circ \Lambda(\text{fold}_F \ R) \supseteq \text{fold}_F (\text{thin } \leq \circ \Lambda(R \circ F \in))$$

is to see the *thin* \leq on the left-hand side as relaxing the constraints on the results returned: for each element returned by fold_F , we allow the algorithm to instead return something it relates to by \succeq (recall that \leq is swapped to \succeq since we take maximisation problems as default in this paper). The right-hand side performs the relaxation inside fold_F , possibly many times, after each invocation of R . The theorem states that the right-hand side returns results accepted by the left-hand side, provided that R is monotonic on \leq , that is, it maps \leq -related inputs to \leq -related outputs. We need transitivity of \leq to guarantee that multiple invocations of *thin* \leq still yields a result acceptable by *thin* \leq on the left-hand side.

Rather than appealing to transitivity, we could instead generalise the theorem to three orderings $\leq^{*\delta}$, $\leq^{\delta 1}$, and $\leq^{+\delta}$. Instead

³ In this paper we sometimes denote multiplication by juxtaposition and use (\cdot) when there could be confusion with function application.

of monotonicity on \preceq , we demand that R be a homomorphism between $\preceq^{*\delta}$ and $\preceq^{+\delta}$:

$$R \circ F \preceq^{*\delta} \subseteq \preceq^{+\delta} \circ R,$$

that is, R takes $\preceq^{*\delta}$ -related input to $\preceq^{+\delta}$ -related results. In addition, we require that $\preceq^{+\delta}$ -related results, after being relaxed by \preceq^{δ_1} , become $\preceq^{*\delta}$ -related again. That is, $\preceq^{+\delta} \circ \preceq^{\delta_1} \subseteq \preceq^{*\delta}$. Given these assumptions, we claim that:

Theorem 2. *For all relations R , $\preceq^{*\delta}$, \preceq^{δ_1} , and $\preceq^{+\delta}$ satisfying $\preceq^{+\delta} \circ \preceq^{\delta_1} \subseteq \preceq^{*\delta}$ and $R \circ F \preceq^{*\delta} \subseteq \preceq^{+\delta} \circ R$, we have*

$$\text{thin } \preceq^{*\delta} \circ \Lambda \text{fold}_F R \supseteq \text{fold}_F (\text{thin } \preceq^{\delta_1} \circ \Lambda(R \circ F)).$$

Proof of the theorem is very similar to that of Theorem 1.

What is this theorem good for? Among other potential uses, it turns out that the generalised theorem plays a central role in proving correctness of fold-based FPTAS algorithms. With some careful design, $\text{thin } \preceq^{*\delta}$ can be used to denote approximation within an acceptable range of error, measured in terms of the size of the input. The theorem allows the relaxation phase to be performed, by $\text{thin } \preceq^{\delta_1}$, after processing each piece of the input in fold_F . That R being homomorphic and $\preceq^{+\delta} \circ \preceq^{\delta_1} \subseteq \preceq^{*\delta}$ control the propagation of error. It may appear to be a hard task to invent three relations, yet in practice they are often lifted from one relation. We will see the details in the next section.

4. Fully Polynomial-Time Approximation Scheme

Recall that an optimisation problem with respect to an objective function f is specified by:

$$\text{opt} = \max \leq_f \circ \Lambda \text{solutions}.$$

The usual definition of a ρ -approximation for opt is an algorithm that, for all x, y such that $y \rightsquigarrow \text{opt } x$, delivers some $z \rightsquigarrow \text{solutions } x$ such that

$$(f y / f z) \uparrow (f z / f y) \leq \rho. \quad (5)$$

where \uparrow stands for the binary maximum operator.

The number $\rho :: \mathbb{R}$ is called the *performance ratio*. The definition intends to apply to both maximisation and minimisation problems: by taking the maximum of $f y / f z$ and $f z / f y$, the left hand side of (5) always reduces to the larger value divided by the smaller value.⁴ We may thus always take $\rho \geq 1$.⁵ Some problems have approximation algorithms with a small performance ratio, while some do not possess ρ -approximation algorithms for any constant ρ , unless $P = NP$. The *fully polynomial time approximation scheme* (FPTAS) is a class of algorithms that is able to ρ -approximate the optimal solution for all $\rho \geq 1$, with running time bounded by a polynomial of the size of the problem and $1/(\rho - 1)$.

Let us come up with a definition of ρ -approximation in terms of \max and \min . Define for all ρ

$$y \leq_f^\rho z \equiv f y \leq \rho \cdot f z.$$

When $y \leq_f^\rho z$ we say that z *subsumes* y by ratio ρ : z may or may not have a f -value larger than that of y but the f -value of z is at least large enough that $f y / f z \leq \rho$. Note that \leq_f^ρ is *not* transitive.

The relation apx maps the input to some solution z that subsumes any other solutions y , in particular those maximal solutions, by ratio ρ :

$$\text{apx} = \max \leq_f^\rho \circ \Lambda \text{solutions}.$$

⁴ Woeginger [16] used a different definition of approximation for maximisation problems that is implied by the definition here.

⁵ FPTASs are sometimes discussed only with $\rho > 1$. Most theorems in this paper also hold for $\rho = 1$ so we include this case by default.

That is, apx specifies ρ -approximations of opt .

Similarly, ρ -approximations of a minimisation problem $\text{opt} = \min \leq_f \circ \Lambda \text{solutions}$ can be specified by $\text{apx} = \min \leq_f^\rho \circ \Lambda \text{solutions}$, a relation that guarantees to return some y such that $(f y / f z) \leq \rho$ for any solution z .

We consider problems for which solutions can be specified as a fold, namely $\text{solutions} = \text{fold}_F R$. The purpose of this section is to establish conditions (see Theorem 3) under which

$$\max \leq_f^\rho \circ \Lambda(\text{fold}_F R)$$

can be refined to

$$\max \leq_f^{\delta^n} \circ \text{fold}_F (\text{thin } \preceq \circ \Lambda(R \circ F)), \quad (6)$$

where $\leq_f^{\delta^n}$, a sub-ordering of \leq_f^ρ , is determined by ρ and will be defined later, while the programmer, like in the case for the ordinary Thinning Theorem, has to invent \preceq and show that it satisfies the constraints demanded by Theorem 3 for all $\rho \geq 1$. The ordering \preceq can then be used to thin the set of solutions, and the resulting set is guaranteed to contain a solution that subsumes the optimal solution by ratio ρ .

The development proceeds in several steps. In Section 4.1 we talk about some theories allowing us to split the approximation process into each stage of fold_F . Section 4.2 formalises the idea in the previous section in a generic setting, while in Section 4.3 we make use of Theorem 2 to promote thin into each stage of the approximation algorithm.

4.1 Stepwise Approximation

The algorithm (6) computes, in each step of the fold, approximate solutions that are slightly worse than the optimal — solutions with some error. As the algorithm proceeds, the error magnifies. We therefore have to be sure that, a numbers of steps later, the accumulated error is still within the given performance ratio.

The user picks a desired approximation ratio $\rho = 1 + \epsilon$, with $\epsilon \geq 0$ a real number. The following theorem was given as Proposition 4.1.(ii) by Woeginger [16]:

Lemma 1. *For all $0 \leq a \leq 1$ and $m \geq 1$, $(1 + a/m)^m \leq 1 + 2a$ holds.*

Let n be the number of steps in which the algorithm produces an approximate solution. By Lemma 1 we have:

$$f y \leq (1 + \epsilon) \cdot f z \Leftarrow f y \leq (1 + \epsilon/2n)^n \cdot f z. \quad (7)$$

That is, if we ensure that each step of (6) computes solutions that subsume the optimal solution by $\delta = 1 + \epsilon/2n$, we may guarantee that n steps later, the remaining solutions subsume the optimal solution by δ^n , which is still bounded by $1 + \epsilon = \rho$. From now on we will denote this “stepwise ratio” $1 + \epsilon/2n$ by δ .

Remark For some objective functions f in Woeginger’s examples [16] (mainly non-linear functions involving, for example, sum of squares of some values) we have to consider a more general case:

$$f y \leq (1 + \epsilon) \cdot f z \Leftarrow f y \leq (1 + \epsilon/2gn)^{gn} \cdot f z,$$

for some constant g , and ensure that each stage incurs an error within $(1 + \epsilon/2gn)^g$. For the examples in this paper it is sufficient to take $g = 1$. The theorems easily extend to the general case. **End of Remark.**

4.2 Size Annotations

The idea in Section 4.1 has to be formalised and recast to a datatype generic setting. In this section we annotate solutions with the number of steps it took to compute them, and thereby keep a record of the range of error allowed. The annotations will eventually be removed from the resulting approximation algorithm, but are used for reasoning about its correctness.

We assume a generic function computing the size for an inductive datatype defined as the least fixed-points of functor F :

$$size_F = fold_F suc_F,$$

with $suc_F :: \mathbb{F}\mathbb{N} \rightarrow \mathbb{N}$. An obvious definition for $FX = 1 + (A \times X)$, for example, is $suc_F (inl ()) = 0$ and $suc_F (inr (a, n)) = 1 + n$; $size_F$ for lists therefore coincides with $length$. For $FX = A + (X \times X)$ we take $suc_F (inl a) = 1$ and $suc_F (inr (n_1, n_2)) = n_1 + n_2 + 1$.

We consider $solutions :: A \leftarrow T$ expressed by a fold, namely $solutions = fold_F R$. The relation $\langle solutions, size_F \rangle :: (A \times \mathbb{N}) \leftarrow T$ maps the input to solutions paired with the size of the input. Define an operator \dashv lifting a relation $X :: A \leftarrow FA$:

$$\begin{aligned} X^\dagger &:: (A \times \mathbb{N}) \leftarrow F(A \times \mathbb{N}) \\ X^\dagger &= \langle X \circ Ffst, suc_F \circ Fsnd \rangle. \end{aligned}$$

Standard transformation (e.g. fold fusion with id) then gives us:

$$\langle solutions, size_F \rangle = fold_F R^\dagger,$$

that is, each solution computed in each stage of $fold_F$ is paired with the current step count. Taking lists for example, if $solutions = foldr S s$, we have $\langle solutions, length \rangle = foldr S' s'$, where $(y, n+1) \leftarrow S' (a, (x, n)) \equiv y \leftarrow S (a, x)$ and $s' = \{(y, 0) \mid y \in s\}$.

An *endo-relation* on A is a relation having type $A \leftarrow A$, which we will abbreviate to $endo A$. For the purpose of precisely describing our theorems later, we need several ways to lift $endo A$ to $endo (A \times \mathbb{N})$. Given $\sqsubseteq :: endo A$, we define $\sqsubseteq^1 :: endo (A \times \mathbb{N})$:

$$(a_1, n) \sqsubseteq^1 (a_2, n) \equiv a_1 \sqsubseteq a_2.$$

Operator \sqsubseteq^1 binds very loosely. For example, $\sqsubseteq_f^{\rho_1} = (\sqsubseteq_f^\rho)^1$.

Some endo-relation take a real number parameter. For example, \sqsubseteq_f^δ can be seen as a function $\sqsubseteq_f :: \mathbb{R} \rightarrow endo A$. Given $\sqsubseteq :: \mathbb{R} \rightarrow endo A$, the operator \sqsubseteq^* produces a relation $endo (A \times \mathbb{N})$:

$$\begin{aligned} \sqsubseteq^* &:: (\mathbb{R} \rightarrow endo A) \rightarrow \mathbb{R} \rightarrow endo (A \times \mathbb{N}) \\ (a_1, n) \sqsubseteq^{*\delta} (a_2, n) &\equiv a_1 \sqsubseteq^{\delta^n} a_2. \end{aligned}$$

Note that both \sqsubseteq and δ are parameters to \sqsubseteq^* , and \sqsubseteq^{δ^n} is \sqsubseteq given parameter δ^n . For example, given $f :: A \rightarrow B$ and $\delta :: \mathbb{R}$, we have $(a_1, n) \sqsubseteq_f^{*\delta} (a_2, n)$ iff $a_1 \sqsubseteq_f^{\delta^n} a_2$.

Recall that an approximation algorithm is specified by $apx = max \sqsubseteq_f^\rho \circ \Lambda(fold_F R)$. The purpose of the definitions above is to come up with a refinement of apx in terms of $max \sqsubseteq_f^{*\delta}$ where $\delta = 1 + \epsilon/2n$: for input size n , the approximate solution is allowed to subsume other solutions by ratio δ^n .

The refinement can be performed in two steps:

$$\begin{aligned} &y \leftarrow (max \sqsubseteq_f^\rho \circ \Lambda(fold_F R)) x \\ &\equiv y \leftarrow (fst \circ max \sqsubseteq_f^{\rho_1} \circ \Lambda(fold_F R^\dagger)) x \\ &\Leftarrow \{ \text{let } n = size_F x \text{ and } \delta = 1 + \epsilon/2n \} \\ &y \leftarrow (fst \circ max \sqsubseteq_f^{*\delta} \circ \Lambda(fold_F R^\dagger)) x. \end{aligned}$$

The first step annotates each solution with the size of the input and compare them by $\sqsubseteq_f^{\rho_1}$. The second step is true because (7) can be rewritten as:

$$(y, n) \sqsubseteq_f^{\rho_1} (z, n) \Leftarrow (y, n) \sqsubseteq_f^{*\delta} (z, n),$$

that is, $\sqsubseteq_f^{\rho_1} \subseteq \sqsubseteq_f^{*\delta}$. In conclusion, $fst \circ max \sqsubseteq_f^{*\delta} \circ \Lambda(fold_F R^\dagger)$ is a valid refinement of apx : for y to subsume all other solutions of x by ρ , it is sufficient if y subsumes them by δ^n .

4.3 Generalised Thinning

The next step is to distribute the allowed error δ^n into the n steps of the fold. To formalise that, we define, for an endo-relation \sqsubseteq^δ

parameterised by a real number δ , yet another lifting:

$$(a_1, n+1) \sqsubseteq^{+\delta} (a_2, n+1) \equiv a_1 \sqsubseteq^{\delta^n} b_2.$$

For an intuition, consider lifting \sqsubseteq_f^δ by the three operators we have introduced so far. The ordering $\sqsubseteq_f^{+\delta}$ is ‘‘one step behind’’ $\sqsubseteq_f^{*\delta}$: a_1 and a_2 are annotated by $n+1$ but a_2 subsumes a_1 only by δ^n . Note that we have for all $\delta \geq 1$:

$$\begin{aligned} (a_1, n+1) \sqsubseteq_f^{+\delta} (a_2, n+1) \wedge a_2 \sqsubseteq_f^\delta a_3 \\ \Rightarrow (a_1, n+1) \sqsubseteq_f^{*\delta} (a_3, n+1). \end{aligned}$$

A fancy way to write it is $\sqsubseteq_f^{+\delta} \circ \sqsubseteq_f^{\delta^1} \subseteq \sqsubseteq_f^{*\delta}$ for all $\delta \geq 1$.

For many problems, however, it will not be \sqsubseteq_f^δ that we want to lift. Like in Section 2.3, we often attempt to find a sub-ordering \sqsubseteq^δ of $\sqsubseteq_f^{*\delta}$ that satisfy the monotonicity/homomorphic condition that allows us to perform thinning within $fold_F$. The property $\sqsubseteq^{+\delta} \circ \sqsubseteq^{\delta^1} \subseteq \sqsubseteq^{*\delta}$ thus becomes a requirement on \sqsubseteq .

Putting all the pieces together, we obtain the following theorem:

Theorem 3. *Let R be a relation of type $A \leftarrow FA$, $f :: A \rightarrow \mathbb{R}$ an objective function, and $\rho = 1 + \epsilon \geq 1$ a performance ratio. If we find $\sqsubseteq :: \mathbb{R} \rightarrow endo A$ such that for all $\delta \geq 1$:*

1. $\sqsubseteq^{*\delta} \subseteq \sqsubseteq_f^{*\delta}$,
2. $\sqsubseteq^{+\delta} \circ \sqsubseteq^{\delta^1} \subseteq \sqsubseteq^{*\delta}$,
3. $R^\dagger \circ F \sqsubseteq^{*\delta} \subseteq \sqsubseteq^{+\delta} \circ R^\dagger$,

then we have, for $n = size_F x$ and $\delta = 1 + \epsilon/2n$:

$$\begin{aligned} &y \leftarrow (max \sqsubseteq_f^\rho \circ \Lambda(fold_F R)) x \\ &\Leftarrow y \leftarrow (max \sqsubseteq_f^{\delta^n} \circ fold_F (thin \sqsubseteq^\delta \circ \Lambda(R \circ F\epsilon))) x. \end{aligned}$$

Proof.

$$\begin{aligned} &y \leftarrow (max \sqsubseteq_f^\rho \circ \Lambda(fold_F R)) x \\ &\Leftarrow \{ \text{Section 4.2} \} \\ &y \leftarrow (fst \circ max \sqsubseteq_f^{\delta^*} \circ \Lambda(fold_F R^\dagger)) x \\ &\Leftarrow \{ \text{condition 1: } \sqsubseteq_f^{\delta^*} \supseteq \sqsubseteq^{*\delta} \} \\ &y \leftarrow (fst \circ max \sqsubseteq_f^{\delta^*} \circ thin \sqsubseteq^{*\delta} \circ \Lambda(fold_F R^\dagger)) x \\ &\Leftarrow \{ \text{Theorem 2 (using conditions 2 and 3)} \} \\ &y \leftarrow (fst \circ max \sqsubseteq_f^{\delta^*} \circ fold_F (thin \sqsubseteq^{\delta^1} \circ \Lambda(R^\dagger \circ F\epsilon))) x \\ &\Leftarrow \{ \text{standard transformation} \} \\ &y \leftarrow (max \sqsubseteq_f^{\delta^n} \circ fold_F (thin \sqsubseteq^\delta \circ \Lambda(R \circ F\epsilon))) x. \end{aligned}$$

□

Theorem 3 pictures an outline how one might construct an FPTAS. Given the problem specification, we need some \sqsubseteq that satisfies the three conditions for every ρ . Conditions 1 and 2 ensure that \sqsubseteq propagate error in a way that agrees with the problem definition and the size constraints. Condition 3 ensures that R does not magnify the error: on the left-hand side two solutions are related by $\sqsubseteq^{*\delta}$ and on the right-hand side by $\sqsubseteq^{+\delta}$, thus R^\dagger increases the size count but not the error. The error is generated purely from $thin \sqsubseteq^\delta$ in $fold_F$.

It may appear to be rather difficult to come up with a \sqsubseteq whose liftings satisfy so many conditions together. As we will show with our examples, however, \sqsubseteq is often a natural extension of the ordering we use for the thinning phase of the exact algorithm.

Efficiency Usually, f and R are polynomial-time computations, and $thin \sqsubseteq^\delta$ can be implemented to run in time bounded by a polynomial of the size of the set it processes. Time complexity of the implementation thus mainly depend on the size of sets of

solutions in each step. A typical approach to obtain an upper bound of the complexity is to estimate the maximum size of subsets of the range of R whose elements are incomparable under \preceq^δ and see whether it is a polynomial of n and $1/\epsilon$.

5. Applications

In this section we derive FPTASs for three NP-hard problems. We begin with resuming our discussion on 0-1 knapsack, followed by solving an optimal scheduling problem proposed by Woeginger [16]. To demonstrate an application on trees, we construct an FPTAS for a tree partitioning problem proposed by Ito et al [13].

For the latter two problems we will first come up with specifications of the optimal versions of the problem and derive thinning algorithms by Theorem 1. We will refer to the algorithms so developed as the “exact” algorithms. The approximation algorithms will be developed using Theorem 3 based on the exact algorithms.

5.1 0-1 Knapsack

Recall that the 0-1 knapsack problem can be specified as

$$\text{knapsack} = \max \leq_v \circ \text{sublist},$$

where *sublist* can be defined by $\Lambda(\text{foldr}(\text{scons} \cup \text{snd}) \{[]\})$, with $(a : x) \leftarrow \text{scons}(a, x) \equiv \text{weight}(a : x) \leq W$. The subrelation of \leq_v we chose for thinning is $x \preceq y \equiv x \leq_v y \wedge x \geq_w y$.

With a user-given performance ratio ρ , the approximate version of 0-1 knapsack can be specified by:

$$\text{knapsack-apx} = \max \leq_v^\rho \circ \text{sublist}.$$

To apply Theorem 3 we need to invent an ordering that satisfies its preconditions, and it turns out that one may simply extend \preceq with an additional parameter δ :

$$x \preceq^\delta y \equiv x \leq_v \delta \cdot y \wedge x \geq_w y.$$

The rationale relaxing \leq_v but not \geq_w is that accuracy on value can be sacrificed for efficiency, while the constraint on weight is a safety condition we cannot compromise.

We have to check that \preceq^δ is a valid choice for Theorem 3. Since $x \preceq^\delta y$ implies $x \leq_v^\delta y$, we also have that $(x, n) \preceq^{*\delta} (y, n)$ implies $(x, n) \leq_v^{*\delta} (y, n)$ and thus condition 1 of Theorem 3 is satisfied. Condition 2 follows from the definition of \preceq^δ .

To check that $x \preceq^\delta y$ satisfies condition 3, we will only look at the non-empty case: let $R = \text{scons} \cup \text{snd}$, we need that for $a :: \text{Item}$, $x_1, y_1, y_2 :: [\text{Item}]$, $n :: \mathbb{N}$, and $\delta :: \mathbb{R} \geq 1$:

$$\begin{aligned} (x_1, n + 1) \leftarrow R^\dagger(a, (y_1, n)) \wedge (y_1, n) \preceq^{*\delta} (y_2, n) \Rightarrow \\ (\exists x_2 : (x_1, n + 1) \preceq^{+\delta} (x_2, n + 1) \wedge \\ (x_2, n + 1) \leftarrow R^\dagger(a, (y_2, n))). \end{aligned}$$

Expanding the liftings, we get

$$\begin{aligned} x_1 \leftarrow R(a, y_1) \wedge y_1 \preceq^{\delta^n} y_2 \Rightarrow \\ (\exists x_2 : x_1 \preceq^{\delta^n} x_2 \wedge x_2 \leftarrow R(a, y_2)). \end{aligned}$$

If x_1 on the left-hand side of the implication is a result of *snd* in R , that is, $x_1 = y_1$, on the right-hand side we can simply take $x_2 = y_2$ and make the implication hold. If x_1 is a result of *scons*, that is, $x_1 = a : y_1$ and $\text{weight } x_1 \leq W$, we claim that $x_2 = a : y_2$ is a valid choice. We then have to show that $a : y_1 \preceq^{\delta^n} a : y_2$. It is easy to see that $a : y_1 \geq_w a : y_2$, since $y_1 \preceq^{\delta^n} y_2$ implies $y_1 \geq_w y_2$. Note that it also implies $\text{weight}(a : y_2) \leq W$. To show that $a : y_1 \leq_v^{\delta^n} a : y_2$, we reason:

$$\begin{aligned} \text{value}(a : y_1) \\ = \text{value } a + \text{value } y_1 \end{aligned}$$

$$\begin{aligned} \text{bump } \delta [] &= [] \\ \text{bump } \delta [x] &= [x] \\ \text{bump } \delta (x : y : xs) & \begin{cases} x \leq_w y \wedge x \geq_v y = \text{bump } \delta (x : xs) \\ \overline{x \geq_w y \wedge x =_v y} = \text{bump } \delta (y : xs) \\ \overline{x \geq_w y \wedge x \leq_v^\delta y} = \text{bump}' \delta (y : xs) \\ \overline{x \geq_w y \wedge x \geq_v y} = x : \text{bump } \delta (y : xs) \end{cases} \end{aligned}$$

$$\begin{aligned} \text{bump}' \delta [] &= [] \\ \text{bump}' \delta [x] &= [x] \\ \text{bump}' \delta (x : y : xs) & \begin{cases} x \leq_w y \wedge x \geq_v y = \text{bump}' \delta (x : xs) \\ \overline{x \geq_w y \wedge x =_v y} = \text{bump}' \delta (y : xs) \\ \overline{x \geq_w y \wedge x \geq_v y} = x : \text{bump } \delta (y : xs) \end{cases} \end{aligned}$$

Figure 2. An approximate version of *bump*.

$$\begin{aligned} &\leq \{ \text{since } y_1 \preceq^{\delta^n} y_2 \Rightarrow y_1 \leq^{\delta^n} y_2 \} \\ &\quad \text{value } a + \delta^n \cdot \text{value } y_2 \\ &\leq \{ \text{since } \delta \geq 1 \} \\ &\quad \delta^n \cdot \text{value } a + \delta^n \cdot \text{value } y_2 \\ &= \delta^n \cdot \text{value}(a : y_2). \end{aligned}$$

What we have shown is that adding an item a to y_1 and y_2 does not magnify the error.

By Theorem 3, we have $y \leftarrow \text{knapsack-apx } x$ if

$$y \leftarrow (\max \leq_v^{\delta^n} \circ \text{foldr}(\text{thin } \preceq^\delta \circ \Lambda S) \{[]\}) x,$$

where $S = (\text{scons} \cup \text{snd}) \circ \text{distr}$, $n = \text{length } x$ and $\delta = 1 + \epsilon/2n$.

Refining to Functions The resulting algorithm is rather similar to the exact algorithm in in Section 2.4:

$$\begin{aligned} \text{knapsack } x &= \text{head}(\text{foldr } \text{step} \{[]\} x) \\ \text{where } n &= \text{length } x \\ \delta &= 1 + \epsilon/2n \\ \text{step } a \text{ } xs &= \text{bump } \delta (\text{merge value } xs \\ &\quad [a : x \mid x \leftarrow xs, \text{weight}(a : x) \leq W]). \end{aligned}$$

It also maintains a list of packings sorted by their values, using the same *merge* as defined in Figure 1 but *bump* defined in Figure 2. Let xs' be the result of *merge* in function *step*.

The purpose of *bump* δ is to trim xs' as much as possible while ensuring that for every x in xs' , there exists some y in the trimmed list such that $x \leq_v^\delta y$. Comparing with Figure 1, the *bump* here has an additional case allowing x to be removed when $x \leq_v^\delta y$. After the removal, however, we shall not remove y in favour of some z even if $y \leq_v^\delta z$, since \leq_v^δ is not transitive, and z could be so small that $x \leq_v^\delta z$ does not hold. Thus we call an auxiliary function *bump'*, which passes the control back to *bump* when we are sure that y will be kept.

Efficiency Consider a list of packings $[x_0, x_1, \dots]$ returned by *bump* δ . Each packing in the list is at least δ -times as large as the previous one, i.e., $v x_0 > \delta \cdot (v x_1)$, $v x_1 > \delta \cdot (v x_2)$, and so on. Therefore, the length of the list is bounded by $\log_\delta V$ where $V = \text{sum}(\text{map value } x)$. Note that $\log_\delta V$ is polynomial to m , n , and $1/\epsilon$, where m is the maximum number of bits needed to represent the values:

$$\begin{aligned} &\log_\delta V \\ &\leq \{ V \text{ should be expressed in } m \text{ bits} \} \\ &\quad \log_\delta(2^m) \\ &= \{ \text{definition of } \delta \} \\ &\quad m \log_{1+\epsilon/2n} 2 \end{aligned}$$

$$\leq \{ 2 \leq (1 + 1/k)^k \text{ if } k \geq 1 \}$$

$$m \log_{1+\epsilon/2n}((1 + \epsilon/2n)^{2n/\epsilon})$$

$$= 2mn/\epsilon.$$

Thus the maximal number of solutions in each step of *foldr* is $O(mn/\epsilon)$, and the time complexity of the program is $O(mn^2/\epsilon)$. Note that the analysis applies to non-integral values and weights.

5.2 Minimising Total Late Work

Let *Job* be a type representing a job, having a processing time and a due time retrievable by *time*, *due* :: *Job* → *Time*, where *Time* is some non-negative real number and the initial time is 0. The input is a list of jobs (more constraints to be given later), and the task is to come up with a job scheduling that minimises the penalty. There is no penalty for jobs finished before their due times. If a job is “partially overdue”, that is, we start processing it before the due time but cannot finish it before the due time, we are penalised by the difference between its finishing time and due time. For a job started after the due time we are penalised by its processing time.

The problem is due to Woeginger [16, Section 8.5], where he used a more complex setting-up involving two orderings, while we need only one relation which, like in the previous section, is a natural extension of the ordering for the exact algorithm.

Specification The specification Woeginger started with already simplified the problem significantly. Given an optimal scheduling, we always get a scheduling that is not worse by swapping two jobs that are not sorted by due times if they are not completely overdue. Thus we may assume that the input is sorted by due times, the more urgent jobs coming first (the left-hand side of the list), and the task is simplified to going through the list and decide, for each job, whether to do it (in the order of the input) or not (that is, put it in a separate pool of jobs that we do not expect to start processing before their due times).

We sort the list in ascending due time and process it from left to right. The relation *split* defined below splits a list *x* into a pair (*y*, *z*) such that *x* is an interleaving of *y* and *z*:

$$\text{split} :: (\text{Job} \times \text{Job}) \leftarrow [\text{Job}]$$

$$\text{split} = \text{foldl} (\text{work} \cup \text{drop}) \{([\], [\])\},$$

where *work* and *drop* are defined by:

$$\text{work} ((y, z), a) = (y + [a], z),$$

$$\text{drop} ((y, z), a) = (y, z + [a]),$$

and *foldl* has type $(B \leftarrow (B \times A)) \rightarrow B \rightarrow (B \leftarrow [A])$. We let the left component of the returned pair denote the jobs we plan to do and the right component the jobs we give up, hence the names *work* and *drop*.

Given a list of jobs performed, the total time it takes to finish them is computed by:

$$\text{totaltime} :: [\text{Job}] \rightarrow \text{Time}$$

$$\text{totaltime} = \text{sum} \circ \text{map time},$$

and the actual finishing time of each job is computed by:

$$\text{fintimes} :: [\text{Job}] \rightarrow [\text{Time}]$$

$$\text{fintimes} = \text{tail} \circ \text{map totaltime} \circ \text{inits}.$$

The function *delays*, defined on a list of jobs, computes how much each job is late (0 if the job finishes in time):

$$\text{delays} :: [\text{Job}] \rightarrow [\text{Time}]$$

$$\text{delays } x = \text{sum} (\text{zipWith } (\ominus) (\text{fintimes } x) (\text{map due } x))$$

where $f \ominus d = (f - d) \uparrow 0$,

where \uparrow stands for binary maximum. The objective function *penalty* is thus defined by:

$$\text{penalty } (x, y) = \text{delays } x + \text{totaltime } y.$$

Some calculation allows us to define *penalty* inductively on the input:

$$\text{penalty } ([], []) = 0$$

$$\text{penalty } (x, y + [a]) = \text{penalty } (x, y) + \text{time } a$$

$$\text{penalty } (x + [a], y) =$$

$$\text{penalty } (x, y) + ((\text{time } a + \text{totaltime } x - \text{due } a) \uparrow 0).$$

The task is to minimise *penalty*:

$$\text{mlw} = \min \leq_p \circ \Lambda \text{split},$$

where *mlw* stands for “minimising late work” and \leq_p abbreviates \leq_{penalty} .

Optimisation Since $\min \leq_p = \max \geq_p$, to turn the specification into a thinning algorithm, we aim to find an ordering $\succeq \subseteq \geq_p$ such that the monotonicity condition $R \circ F \succeq \subseteq \succeq \circ R$ holds. For the case of non-empty list, it simplifies to:

$$zw_1 \leftarrow R (xy_1, a) \wedge xy_1 \succeq xy_2 \Rightarrow$$

$$(\exists zw_2 : zw_1 \succeq zw_2 \wedge zw_2 \leftarrow R (xy_2, a)).$$

where zw_1, xy_1, xy_2 , etc., are pairs of lists of jobs. We cannot take $\succeq = \geq_p$: it could be the case that $(x_1, y_1) \geq_p (x_2, y_2)$ but $(x_2 + [a], y_2)$ incurs a penalty higher than either $(x_1 + [a], y_1)$ or $(x_1, y_1 + [a])$, because x_2 has a longer running time which delays *a* too much.

If we instead define

$$(x_1, y_1) \succeq (x_2, y_2) \equiv (x_1, y_1) \geq_p (x_2, y_2) \wedge$$

$$\text{totaltime } x_1 \geq \text{totaltime } x_2,$$

and assume that $(x_1, y_1) \succeq (x_2, y_2)$, one can easily see, from the inductive characterisation of *penalty*, that $(x_1 + [a], y_1) \succeq (x_2 + [a], y_2)$ and $(x_1, y_1 + [a]) \succeq (x_2, y_2 + [a])$. Therefore \succeq is a relation we may use for thinning.

By Theorem 1 we have:

$$\text{mlw} \supseteq \min \leq_p \circ \text{foldl} (\text{thin } \succeq \circ \Lambda S) \{([\], [\])\},$$

where $S = (\text{work} \cup \text{drop}) \circ \text{distl}$ with $\text{distl} = \Lambda(\in \times \text{id})$. It can be further refined to an algorithm that maintains a list of solutions sorted by ascending penalties and, for each penalty only one solution (x, y) with minimum *totaltime* *x* is kept. The derived program is similar in spirit to that of 0-1 knapsack and omitted here.

Approximation The approximate specification takes a performance ratio ρ :

$$\text{mlw-approx} = \min \leq_p^\rho \circ \Lambda \text{split},$$

To construct an approximation algorithm using Theorem 3, we have to come up with an ordering satisfying its preconditions. It turns out that we can choose a natural extension of \succeq :

$$(x_1, y_1) \succeq^\delta (x_2, y_2) \equiv (x_1, y_1) \geq_p^\delta (x_2, y_2) \wedge$$

$$\text{totaltime } x_1 \geq \text{totaltime } x_2,$$

where $(x_2, y_2) \leq_p^\delta (x_1, y_1)$ if and only if

$$\text{penalty } (x_2, y_2) \leq \delta \cdot \text{penalty } (x_1, y_1).$$

Again, we allow error on penalties but not the total time, since the latter must be accurate to compute correct solutions.

Due to space constraint let us only check whether $R = \text{work} \cup \text{drop}$ satisfies the homomorphic condition 3 in Theorem 3, which simplifies to that for all $zw_1, xy_1, xy_2 :: ([\text{Job}] \times [\text{Job}])$:

$$zw_1 \leftarrow R (xy_1, a) \wedge xy_1 \succeq^{\delta^n} xy_2 \Rightarrow$$

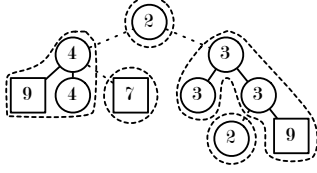


Figure 3. A tree partition with fulfilment 17.

$$(\exists zw_2 : zw_1 \succeq^{\delta^n} zw_2 \wedge zw_2 \leftarrow R(xy_2, a)).$$

Let $(x_1, y_1) \succeq^{\delta} (x_2, y_2)$. Assume that we extended (x_2, y_2) by *work* to $(x_2 + [a], y_2)$. We also extended (x_1, y_1) to $(x_1 + [a], y_1)$. Certainly, $\text{totaltime}(x_1 + [a]) \geq \text{totaltime}(x_2 + [a])$. For the inequality regarding *penalty* we reason (abbreviating *penalty* to p):

$$\begin{aligned} & \delta^n \cdot p(x_1 + [a], y_1) \\ &= \delta^n \cdot ((\text{time } a + \text{totaltime } x_1 - \text{due } a) \uparrow 0) + \delta^n \cdot p(x_1, y_1) \\ &\geq \{ \text{since } \delta^n \cdot p(x_1, y_1) \geq p(x_2, y_2) \} \\ & \delta^n \cdot ((\text{time } a + \text{totaltime } x_1 - \text{due } a) \uparrow 0) + p(x_2, y_2) \\ &\geq \{ \text{since } \text{totaltime } x_1 \geq \text{totaltime } x_2 \} \\ & \delta^n \cdot ((\text{time } a + \text{totaltime } x_2 - \text{due } a) \uparrow 0) + p(x_2, y_2) \\ &> \{ \text{since } \delta > 1 \} \\ & ((\text{time } a + \text{totaltime } x_2 - \text{due } a) \uparrow 0) + p(x_2, y_2) \\ &= p(x_2 + [a], y_2). \end{aligned}$$

The case when (x_2, y_2) is extended by *drop* to $(x_2, y_2 + [a])$ is easier.

By Theorem 3, we have $(z, w) \leftarrow \text{mlw-apx } x$ if

$$(z, w) \leftarrow (\min \leq_p^{\delta^n} \circ \text{foldl}(\text{thin} \succeq^{\delta} \circ \wedge S) \{([\], [\]\}) x,$$

where $S = (\text{work} \cup \text{drop}) \circ \text{distl}$ and $n = \text{length } x$. It can also be refined to an algorithm that maintains a list of solutions sorted by ascending penalties, similar to that for the exact algorithm, but performs more aggressive thinning in a way similar to the approximate algorithm for 0-1 knapsack. Details omitted.

5.3 Partitioning Trees with Supply and Demand

Our last example shall demonstrate the ability of our generic theorems to deal with problems other than those taking lists as inputs. An ideal candidate would be an NP-hard problem defined on trees that has an FPTAS. Due to the difficulty of designing approximation algorithms on complex structures, there have been few FPTAS for NP-hard problems on trees. In this section we will look at a recent instance discussed by Ito et al. [13].

Consider internally labelled rose trees defined by:

data $\text{Tree} = N \text{SD} [\text{Tree}]$,
data $\text{SD} = S \mathbb{N} | D \mathbb{N}$.

Each node is labelled either as $S n$ (a supplier) or $D n$ (a demander) where n is a natural number. Think of each supplier as a power plant, and demanders as consumers connected to the power plant, directly or indirectly, through the edges. The goal is to partition the tree into subtrees such that each tree either contains no supplier at all, or contains exactly one supplier whose amount of electricity generated is large enough to fulfil all demands in the tree. The goal is to maximise fulfilled demands.

An example is shown in Figure 3 where the suppliers are drawn as squares and demanders as circles. The tree is partitioned into 5 parts. The leftmost partition consists of one supplier $S 9$ and two demanders $D 4$, and totally 8 units of demands are fulfilled.

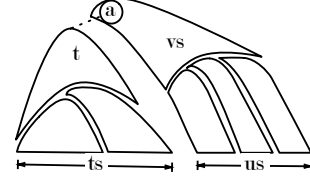


Figure 4. Joining two partitions (t, ts) and $(N a vs, us)$.

The rightmost partition fulfils 9 units of demands from three $D 3$ nodes. The three singleton trees respectively consist of one isolated supplier and two demanders, and nothing is fulfilled. The total fulfilment of this partition is thus 17.

5.3.1 Specification

In the formal specification we will be using *foldT* on *Tree*, whose functional version is discussed in Section 2.2 and recited here:

$$\begin{aligned} \text{foldT} &:: ((SD \times [A]) \rightarrow A) \rightarrow \text{Tree} \rightarrow A \\ \text{foldT } f &(N \text{sd } ts) = f \text{sd} (\text{map } (\text{foldT } f) \text{ts}). \end{aligned}$$

Partitioning We start with formally specifying what it means to partition a tree. A partition is a bag of *Trees*. For space concern, we cut directly to considering how one might compute partitions of a tree in a *foldT*. It will turn out that we have to treat the subtree containing the root node differently from the rest of the partition. Therefore, let us represent a partition by a pair:

$$\text{type Part} = (\text{Tree} \times \wr \text{Tree}\wr),$$

where the first component is the tree containing the root and \wr is the type of bags. The partition in Figure 3 is represented by $(N (D 2) [\], ts)$ where ts is the bag consisting of four other trees.

Consider two partitions (t, ts) and (u, us) , where $u = N a vs$, as shown in Figure 4. There are two ways to build a partition, with root a , containing all the trees. We may connect the dotted line in Figure 4, making t a subtree of a :

$$\text{link } (t, ts) (N a vs, us) = (N a (t : vs), ts \cup us),$$

or we may choose not to connect a and t , and include both t and ts into the bag of trees that do not contain a :

$$\text{sep } (t, ts) (N a vs, us) = (N a vs, \wr t \wr \cup ts \cup us).$$

The relation *part* computes one arbitrary partition of a tree using *foldT*. Each subtree is first recursively partitioned, before the relation *join* goes through the list of partitions and choose to perform either *link* or *sep* in each step:

$$\begin{aligned} \text{part} &:: \text{Part} \leftarrow \text{Tree} \\ \text{part} &= \text{foldT } \text{join} \\ \text{where } \text{join} &:: (SD \times [\text{Part}]) \rightarrow \text{Part} \\ & us \leftarrow \text{join } (a, tss) \equiv \\ & us \leftarrow (\text{foldr } (\text{link} \cup \text{sep}) \{(N a [\], \wr \wr)\}) tss. \end{aligned}$$

The partitions so constructed have not taken feasibility into consideration: a tree may have more than one supplier, or the supply may not be sufficient. We will formalise these concepts in the next few paragraphs.

Evaluation We will define a function *supply* that determines whether a tree has zero, exactly one, or more than one suppliers. Thus we define a datatype enumerating the three possibilities:

$$\text{data Ternary } a = \text{None} | \text{One } a | \text{TooMany}.$$

The function *supply* can then be defined in terms of a *foldT*:

```

supply :: Tree → Ternary ℕ
supply = foldT fetchSupply
  where fetchSupply (S v, ms)
        | all isNone ms = One v
        | otherwise     = TooMany
        fetchSupply (D -, ms) = catT ms,

```

where $catT :: [Ternary\ a] \rightarrow Ternary\ a$ collapses a list of ternary values into one. For example, $catT [None, One\ a, None] = One\ a$ and $catT [One\ a, One\ b] = TooMany$. The definition is routine and omitted.

It is convenient to define predicates *existsS* and *allD* on trees such that *existsS t* is true if the tree has exactly one supplier (that is, *supply t = One s*) and *allD t* if there is no supplier in *t*:

```

existsS t | One s ← supply t = True
           | otherwise       = False,
allD t   | None ← supply t = True
           | otherwise       = False.

```

The definition above makes use of the “pattern guards” syntax [8], implemented in GHC.

The function *demand*, on the other hand, simply returns the sum of all demands:

```

demand :: Tree → ℕ
demand = foldT dsum
  where dsum (D v, vs) = v + sum vs
        dsum (S -, vs) = sum vs.

```

A tree is *feasible* if it either contains no supplier, or exactly one supplier that meets all the demands:

```

feasible :: Tree → Bool
feasible t | One s ← supply t = s ≥ demand t
feasible t | None ← supply t  = True
feasible t | TooMany ← supply t = False.

```

The *fulfilment* of a tree is computed below:

```

fulfilment :: Tree → ℕ
fulfilment t | One s ← supply t,
              d ← demand t, s ≥ d = d
fulfilment t | otherwise           = 0.

```

If there is exactly one supplier in a tree that is able to meet all the demands, the fulfilment is measured in terms of the demands (*not* the supply). Otherwise the fulfilment is zero.

Like before, we overload the function to lists, bags, and partitions of trees in the obvious way:

```

fulfilment :: [Tree] → ℕ
fulfilment = sum ∘ map fulfilment,

```

```

fulfilment :: Part → ℕ
fulfilment (t, ts) = fulfilment t + fulfilment ts.

```

Finally, the *maximal tree partition* problem is specified by:

```
mtp = max ≤ff ∘ Λ(feasible? ∘ part),
```

where $≤_{ff}$ abbreviates $≤_{fulfilment}$.

Build Feasible Partitions by a Fold Again, to apply the two thinning theorems we have to turn *feasible? ∘ parts* into a *foldT*. By fold-fusion, we may express generation of feasible partitions in a *foldT*:

```

fepart :: Part ← Tree
fepart = foldT fejoin
  where fejoin :: (SD × [Part]) → Part
        us ↔ fejoin (a, tss) ≡
        us ↔ (foldr (felink ∪ sep) {(N a [], [])} tss.

```

where $felink = (feasible \circ fst)? \circ link$.

5.3.2 Optimisation

Our goal is to derive an FPTAS solving the maximal tree partition problem, but it would be rather difficult to do so directly from the specification. Like the previous examples, it would be much easier to first come up with a thinning algorithm solving *mtp*, and use that as a hint to derive an FPTAS.

To apply the Theorem 1, we wish to come up with an ordering $≤$ that is a sub-ordering of $≤_{ff}$ on which *fejoin* is monotonic.

It is a common trick to invent suitable sub-orderings by comparing only those solutions belonging to the same “class” in the relation generating all solutions. Since *fejoin*, to generate feasible partitions, distinguishes those partitions whose root tree has one supplier from those having none, we will also pick a sub-ordering this way.

Given a tree with one supplier, its *margin* is the amount of electricity the supplier can still provide after meeting all the demands in the tree:

```

margin :: Tree → ℕ
margin t | One s ← supply t = s - demand t.

```

Consider two partitions such that $(t, ts) ≤_{ff} (u, us)$. If *t* and *u* both contain a supplier, a reasonable guess is that (t, ts) is not worth keeping if $margin\ t ≤ margin\ u$: we can prove that however we extend (t, ts) , we can extend (u, us) the same way and get a partition that is at least as good. Thus we define:

$$(t, ts) ≤_s (u, us) ≡ existsS\ t \wedge existsS\ u \wedge (t, ts) ≤_{ff} (u, us) \wedge t ≤_m u,$$

where $t ≤_m u$ if $margin\ t ≤ margin\ u$.

The case for $allD\ t \wedge allD\ u$ is a little bit trickier. One might guess that (t, ts) can be dropped in favour of (u, us) if

$$(t, ts) ≤_{ff} (u, us) \wedge t ≥_d u,$$

where $t ≥_d u$ if $demand\ t ≥ demand\ u$, ensuring that if some supplier is powerful enough to fulfil *t* it is able to fulfil *u* as well. However, notice that in $(t, ts) ≤_{ff} (u, us)$, both *t* and *u*, not having a supplier, have fulfilment value zero. With $t ≥_d u$, it could be the case that once *t* is fulfilled, the total fulfilment of (t, ts) becomes larger than that of (u, us) .

The lesson is that when we compare (t, ts) and (u, us) (with $allD\ t$ and $allD\ u$), we have to assume that the demands of *t* and *u* are fulfilled already. Define:

$$expectancy(t, ts) = demand\ t + fulfilment\ ts,$$

which denotes the potential fulfilment a partition could have if the root tree can be fully supplied. The ordering $≤_d$ is the ordering we will use to compare partitions whose root trees contain no supplier:

$$(t, ts) ≤_d (u, us) ≡ allD\ t \wedge allD\ u \wedge (t, ts) ≤_{ex} (u, us) \wedge t ≥_d u.$$

where *ex* abbreviates *expectancy*.

It is crucial that $(t, ts) ≤_d (u, us)$ implies $(t, ts) ≤_{ff} (u, us)$, that is, $≤_d$ is still a sub-ordering of $≤_{ff}$, which we leave for the reader to verify.

Let $≤ = ≤_s \cup ≤_d$. We claim that *fejoin* is monotonic on $≤$. By expanding the definition of *fejoin*, we know that its monotonicity is implied by monotonicity of *felink* \cup *sep*, that is, for all $tts_1, uus_i, vvs_i :: Part$ (where $i \in \{1, 2\}$) we have:

$$tts_1 \leftrightarrow S(uus_1, vvs_1) \wedge uus_1 \leq uus_2 \wedge vvs_1 \leq vvs_2 \Rightarrow (\exists tts_2 : tts_1 \leq tts_2 \wedge tts_2 \leftrightarrow S(uus_2, vvs_2)),$$

where $S = felink \cup sep$. The property can be proved by some case analysis. Since it is a simplified version of the homomorphic

property we are going to prove for the approximate algorithm, we will omit its proof here.

5.3.3 Approximation

Given a performance ratio ρ , a specification of the approximate maximal tree partition problem is given by:

$$mtp\text{-}apx = \max \leq_{\text{ff}}^{\rho} \circ \Lambda(\text{feasible?} \circ \text{parts}).$$

Now that we have derived a thinning strategy solving mtp , we may attempt to solve $mtp\text{-}apx$ by extending \preceq in the natural way: let $\preceq^{\delta} = \preceq_s^{\delta} \cup \preceq_d^{\delta}$, where \preceq_s^{δ} and \preceq_d^{δ} are defined by:

$$(t, ts) \preceq_s^{\delta} (u, us) \equiv \text{exists}S t \wedge \text{exists}S u \wedge \\ (t, ts) \leq_{\text{ff}}^{\delta} (u, us) \wedge t \leq_m u,$$

$$(t, ts) \preceq_d^{\delta} (u, us) \equiv \text{all}D t \wedge \text{all}D u \wedge \\ (t, ts) \leq_{\text{ex}}^{\delta} (u, us) \wedge t \geq_d u.$$

Like before, we relax the orderings comparing *fulfilment* and *expectancy*, while keeping comparisons on *margin* and *demand* strict, since the latter are safety conditions ensuring the feasibility of partitions we generate.

The homomorphic condition in Theorem 3 we need to show is that for all $tts_1, uus_i, vvs_i :: \text{Part}$ (where $i \in \{1, 2\}$) we have:

$$tts_1 \leftarrow S (uus_1, vvs_1) \wedge uus_1 \preceq_s^{\delta^m} uus_2 \wedge vvs_1 \preceq_s^{\delta^n} vvs_2 \\ \Rightarrow (\exists tts_2 : tts_1 \preceq_s^{\delta^{m+n}} tts_2 \wedge tts_2 \leftarrow S (uus_2, vvs_2)),$$

where $S = \text{felink} \cup \text{sep}$. Notice the superscript: if uus_2 subsumes uus_1 by δ^m and vvs_2 subsumes vvs_1 by δ^n , we want tts_2 to subsume tts_1 by δ^{m+n} .

The proof proceeds by case analysis on uus_i and vvs_i . Let $uus_i = (u_i, us_i)$ and similarly for tts_i and vvs_i , for $i \in \{1, 2\}$. Here we will only look at the most interesting case: *exists* $S u_i$ and *all* $D v_i$. We further distinguish between two possible sub-cases:

1. tts_1 is generated by *sep*, that is, $tts_1 = (v_1, \{u_1\} \cup us_1 \cup vs_1)$. For this case also pick $tts_2 = (v_2, \{u_2\} \cup us_2 \cup vs_2)$. The task is to show that $tts_1 \preceq_d^{\delta^{m+n}} tts_2$.

Immediately we have $v_1 \geq_d v_2$ because $vvs_1 \preceq_d^{\delta^n} vvs_2$. To show that $tts_1 \leq_{\text{ex}}^{\delta^{m+n}} tts_2$, we reason:

$$\text{demand } v_1 + \text{ff } u_1 + \text{ff } us_1 + \text{ff } vs_1 \\ \leq \{ \text{since } uus_1 \preceq_s^{\delta^m} uus_2 \text{ and thus } uus_1 \leq_{\text{ff}}^{\delta^m} uus_2 \} \\ \text{demand } v_1 + \delta^m \cdot \text{ff } u_2 + \delta^m \cdot \text{ff } us_2 + \text{ff } vs_1 \\ \leq \{ \text{since } vvs_1 \preceq_d^{\delta^n} vvs_2 \text{ and thus } vvs_1 \leq_{\text{ex}}^{\delta^n} vvs_2 \} \\ \delta^n \cdot \text{demand } v_2 + \delta^m \cdot \text{ff } u_2 + \delta^m \cdot \text{ff } us_2 + \delta^n \cdot \text{ff } vs_2 \\ \leq \{ \text{since } \delta \geq 1 \} \\ \delta^{m+n} \cdot (\text{demand } v_2 + \text{ff } u_2 + \text{ff } us_2 + \text{ff } vs_2).$$

2. tts_1 is generated by *felink*, that is, v_1 can be decomposed into $N a ws_1$, and $tts_1 = (N a (u_1 : ws_1), us_1 \cup vs_1)$. For this partition to be feasible, we must have *margin* $u_1 \geq \text{demand } v_1$.

For this case we also decompose v_2 into $N a ws_2$, and pick $tts_2 = (N a (u_2 : ws_2), us_2 \cup vs_2)$.

We have to show that $N a (u_2 : ws_2)$ is feasible, that is

$$\text{margin } u_2 \geq \text{demand } v_2.$$

Then we show that $tts_1 \preceq_s^{\delta^{m+n}} tts_2$, one of the conditions is

$$N a (u_1 : ws_1) \leq_m N a (u_2 : ws_2).$$

Both of them are evident because $uus_1 \preceq_s^{\delta^m} uus_2$ and $vvs_1 \preceq_d^{\delta^n} vvs_2$ imply $u_1 \leq_m u_2$ and $v_1 \geq_d v_2$, respectively.

The proof involving approximation is about showing that

$$tts_1 \leq_{\text{ff}}^{\delta^{m+n}} tts_2,$$

for which we reason:

$$\text{fulfilment } tts_1 \\ = \text{ff } u_1 + \text{demand } v_1 + \text{ff } us_1 + \text{ff } vs_1 \\ \leq \{ \text{since } uus_1 \leq_{\text{ff}}^{\delta^m} uus_2 \} \\ \delta^m \cdot \text{ff } u_2 + \text{demand } v_1 + \delta^m \cdot \text{ff } us_2 + \text{ff } vs_1 \\ \leq \{ \text{since } vvs_1 \leq_{\text{ex}}^{\delta^n} vvs_2 \} \\ \delta^m \cdot \text{ff } u_2 + \delta^n \cdot \text{demand } v_2 + \delta^m \cdot \text{ff } us_2 + \delta^n \cdot \text{ff } vs_2 \\ \leq \{ \text{since } \delta \geq 1 \} \\ \delta^{m+n} \cdot (\text{ff } u_2 + \text{demand } v_2 + \text{ff } us_2 + \text{ff } vs_2) \\ = \delta^{m+n} \cdot \text{fulfilment } tts_2.$$

With the monotonicity proved, we may apply Theorem 3 and conclude that we have $(t, ts) \leftarrow mtp\text{-}apx u$ if

$$(t, ts) \leftarrow (\max \leq_{\text{ff}}^{\delta^n} \circ \text{fold}T (\text{thin } \preceq^{\delta} \circ \Lambda S)) u,$$

where $S = \text{fejoin} \circ (\text{id} \times \in \times \in)$ and $n = \text{size}_F u$.

5.3.4 Refining to Function

The functional refinement of *thin* \preceq^{δ} has to discriminate between partitions whose root trees contain suppliers from those do not. Therefore, we represent the set of partitions as a pair

$$(uss, vss) :: ([\text{Part}] \times [\text{Part}]),$$

where uss contains partitions (u, us) such that *exists* $S u$ holds, while vss contains (v, vs) satisfying *all* $D v$.

Similar to the case for 0-1 knapsack, uss is sorted by decreasing *fulfilment* and, for each *fulfilment* we keep only one partition having maximum *margin*. Likewise, vss is sorted by decreasing *expectancy*, and for each *expectancy* we keep one partition having minimum *demand*.

The main algorithm is implemented by a *fold* T returning the pair (uss, vss) :

$$mtp\text{-}apx :: \text{Tree} \rightarrow \text{Part} \\ mtp\text{-}apx = \max \leq_{\text{ff}}^{\rho} \circ \text{cut} \circ \text{fold}T \text{thinjoin}, \\ \text{where } \text{cut} ([], vss) = vss \\ \text{cut } (us : -, vss) = us : vss.$$

To take the maximum of uss we may simply take its head. The list vss , however, is sorted by *expectancy* and we thus have to go through it again, by $\max \leq_{\text{ff}}^{\rho}$, to pick a maximum.

The function *thinjoin* is an implementation of *thin* $\preceq^{\delta} \circ \Lambda S$:

$$\text{thinjoin} :: (SD \times ([\text{Part}] \times [\text{Part}])) \rightarrow ([\text{Part}] \times [\text{Part}]) \\ \text{thinjoin } (Sv, tss) = \text{foldr } \text{step} ([(N (Sv) [], \{ \}), []], tss) \\ \text{thinjoin } (Dv, tss) = \text{foldr } \text{step} ([[], (N (Dv) [], \{ \})], tss),$$

while *step*, of type $(([\text{Part}] \times [\text{Part}]) \times ([\text{Part}] \times [\text{Part}])) \rightarrow ([\text{Part}] \times [\text{Part}])$ is in essence an implementation of *thin* $\preceq^{\delta} \circ \Lambda(\text{felink} \cup \text{sep})$, in which we need list comprehensions to construct the trees. What is important here, however, is that the list uss is generated by *merge fulfilment* and *bump* δ *fulfilment* (\geq_m), while vss by *merge expectancy* and *bump* δ *expectancy* (\leq_d).

Efficiency Let V be the total possible supply. Any candidates taken into our consideration should have *fulfilments* (or *expectations*) that is at most V . Moreover, our implementation of *thin* \preceq^{δ} ensures that the ratio of *fulfilments* (or *expectations*) of any two candidates in the same lists are larger than δ . Therefore, the size of each list is bounded by $\log_{\delta} V$. A reasoning similar to the 0-1

knapsack problem shows that the time complexity of our algorithm is $O(mn^2/\epsilon)$, where m is the number of bits used to input the values of the suppliers.

6. Conclusion and Related Work

We have presented a formulation of fold-based fully polynomial-time approximation schemes, based on a generalisation of the Thinning Theorem. We have thus filled in a missing piece in the jigsaw puzzle of the genealogy of algorithms — while thinning algorithms are generalised from greedy algorithms by employing non-connected preorders, fold-based approximation algorithms can be seen as generalised from thinning by relaxing transitivity to relationships between three relations that can be generated by lifting from one relation.

The formulation also suggests a way to develop FPTAS. Once we know how to solve an optimisation problem by thinning with respect to some ordering \preceq , the relation \preceq^δ we need to derive an approximation algorithm can often be constructed by relaxing the comparisons in \preceq on non-safety related components. It can then be lifted to the three relations demanded by the generalised thinning.

Like the Thinning Theorem, our formulation deals with only fold-based algorithms. It ought to be possible, however, to extend the framework to more general settings in a way similar to how Curtis [6] extended thinning to iteration-based algorithms.

FPTAS and related notions have a long history. Ibarra and Kim [12] discovered an approximation scheme for the 0-1 knapsack problem. Instead of considering maximals on \leq^δ , they multiplied δ to values of items. With several optimisations, the algorithm they proposed has time complexity $O(n \log n + 1/\epsilon^4 \log(1/\epsilon))$. Plenty of approximation algorithms in a similar style have been developed afterwards. Arora [1] compiled a nice summary of approximation problems in his survey of geometric computations.

Some researchers tried to provide characterisations of problems that have FPTAS [2, 14, 16]. Among others, our work is much influenced by Woeginger [16]. Instead of proving the existence of approximation algorithms, Woeginger pointed out the similarity between dynamic programming algorithms and FPTAS and proposed a method of *developing* the latter from the former. Woeginger’s formulation of FPTAS involves a partial order \leq_{dom} , on which the generating relation R must be monotonic (which can be seen as assuming that there exists a thinning algorithm), a preorder \leq_{qua} , and a symmetric relation “[D, Δ]-closeness” denoting that two solutions are internally close enough. The homomorphic condition Woeginger demanded include (denoting [D, Δ]-closeness by Δ):

$$R \circ F(\leq_{qua} \cap \Delta) \subseteq (\leq_{dom} \cup (\leq_{qua} \cap \Delta)) \circ R,$$

which we believe to be unnecessarily complicated. Indeed, among Woeginger’s examples only two problems needed a \leq_{dom} that is not *id*, one of them being the total late work minimisation problem discussed in Section 5.2. Our simpler formulation works for both problems. We believe that the complexity of Woeginger’s formulation is a result of bringing in too early too much detail that can be deferred to later stages of algorithm development.

Halman et al. [10] recently commented on three missing aspects in Woeginger’s work that can be further developed: treelike problems, problem involving convex or monotone functions, and stochastic optimisation problems. Halman et al. dealt with the latter two aspects while we, using a datatype generic approach, extended FPTAS to trees. To the best of our knowledge, our work is the first datatype generic formulation of FPTAS. While our algorithm for tree partitioning in Section 5.3 distinguishes between two classes of solutions, Ito et al. employed three classes (and thus more case analysis). What lead to the difference was perhaps that Ito et al. lacked an expressive language to describe concepts such as *expectancy* and orderings based on them.

Halman et al. [10] extended Woeginger’s formulation to stochastic optimisation, where objective values may include an uncertain factor, and thus the goal is to maximise and minimise the expectancy. It is a possible future work to investigate whether the Thinning Theorem can be further generalised to describe stochastic optimisation.

Acknowledgements The authors would like to thank the anonymous referees for carefully reading the paper and giving many valuable advices.

References

- [1] S. Arora. Approximation schemes for NP-hard geometric optimization problems: a survey. *Mathematical Programming*, 97(1-2):43–69, 2003.
- [2] G. Ausiello, P. Crescenzi, and M. Protasi. Approximate solution of NP optimization problems. *Theoretical Computer Science*, 150(1):1–55, 1995.
- [3] R. S. Bird. A calculus of functions for program derivation. In D. A. Turner, editor, *Research Topics in Functional Programming*, University of Texas at Austin Year of Programming Series, pages 287–308. Addison-Wesley, 1990.
- [4] R. S. Bird and O. de Moor. *Algebra of Programming*. International Series in Computer Science. Prentice Hall, 1997.
- [5] R. S. Bird, J. Gibbons, and S.-C. Mu. Algebraic methods for optimization problems. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, LNCS No. 2297, pages 281–307. Springer-Verlag, 2002.
- [6] S. Curtis. *A Relational Approach to Optimization Problems*. PhD thesis, Oxford University Computing Laboratory, 1995.
- [7] O. de Moor. A generic program for sequential decision processes. In *Programming Languages: Implementations, Logics, and Programs*, LNCS No. 982, pages 1–23. Springer-Verlag, 1995.
- [8] M. Erwig and S. L. Peyton Jones. Pattern guards and transformational patterns. *Electronic Notes in Theoretical Computer Science*, 41(1), 2000.
- [9] M. R. Garey and D. S. Johnson. “Strong” NP-completeness results: motivation, examples, and implications. *Journal of the ACM*, 25(3):499–508, July 1978.
- [10] N. Halman, D. Klabjan, C.-L. Li, J. B. Orlin, and D. Simchi-Levi. Fully polynomial time approximation schemes for stochastic dynamic programs. In *The Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 700–709. SIAM, 2008.
- [11] E. Horowitz and S. Sahni. Exact and approximate algorithms for scheduling nonidentical processors. *Journal of the ACM*, 23(2):317–327, April 1976.
- [12] O. H. Ibarra and C. E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM*, 22(4):463–468, October 1975.
- [13] T. Ito, X. Zhou, and T. Nishizeki. Partitioning trees of supply and demand. *International Journal of Foundations of Computer Science*, 16(4):803–827, August 2005.
- [14] B. Korte and R. Schrader. On the existence of fast approximation schemes. In *Nonlinear Programming*, volume 4, pages 415–437. Academic Press, 1981.
- [15] S.-C. Mu, Y.-H. Lyu, and A. Morihata. <http://www.iis.sinica.edu.tw/~scm/2010/fptas>, 2010.
- [16] G. J. Woeginger. When does a dynamic programming formulation guarantee the existence of an FPTAS? *INFORMS Journal on Computing*, 12:57–75, January 2000.