# On Building Trees with Minimum Height, Relationally

Shin-Cheng Mu
Computing Laboratory, University of Oxford
scm@comlab.ox.ac.uk

**Abstract**

The algebraic style of reasoning about programs has been proposed and studied by computing scientists. We rephrase the old problem of building trees of minimum height as an optimisation problem and apply the greedy theorem to derive a linear time algorithm. To put the problem in the right form, we find it necessary to generalise from functions to relations and make use of the converse of a function theorem to write the inverse of a function as a fold.

## 1 Introduction

Given a list of trees. The task is to combine them into one, retaining the left-to-right order of the trees. How can we combine them so that the height of the resulting tree is as small as possible? The actual contents of the subtrees are not relevant. Therefore we can think of the input as a list of numbers representing the heights of the subtrees. Fig. 1 illustrates one of the best arrangement of subtrees with heights $[2, 9, 8, 3, 6, 9]$, whose height is 11.

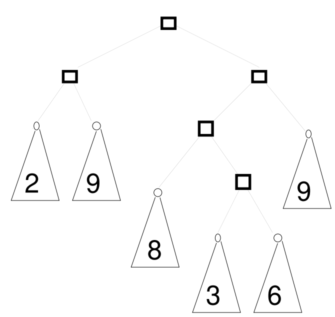


Figure 1: A tree with height 11 built from trees with heights $[2, 9, 8, 3, 6, 9]$

This is actually an instance of the optimal bracketing problem, that is, finding the best way to bracket the expression

$$a_1 \oplus a_2 \oplus a_3 \oplus \ldots a_n$$

in such a way that the value of the expression is minimal. For this problem, the cost function is $a \oplus b = (max\ a\ b) + 1$. The optimal bracketing problem is usually solved via a dynamic programming approach, taking cubic time. A linear-time algorithm for this particular problem, however, has been studied in [1].

This paper is about how this problem can be expressed as an optimisation problem of the following form, which has been extensively studied in [2]:

$$min\ (\unlhd) \cdot \Lambda(foldrn\ step\ base)$$

and how known theories can be applied to derive algorithms to the problem. In the above specification, the function *foldrn* is the fold for non-empty lists:

$$\begin{array}{lcl} foldrn\ step\ base\ [x] & = & base\ x \\ foldrn\ step\ base\ (x : xs) & = & step\ x\ (foldrn\ step\ base\ xs) \end{array}$$

Later in this paper, however, we will generalise *foldrn step base* to be a *relation*. It folds over the input data to generate an arbitrary solution to the optimisation problem. The $\Lambda$ operator collects them all. An optimal one with respect to preorder $(\unlhd)$ is then chosen by $min\ (\unlhd)$. These concepts will be explained more precisely in the following sections.

# 2 Preliminaries

## 2.1 Relations

During the last decade, the computer scientists gradually discovered the benefit for program derivation to generalise from functions to relations. For our purpose, it suffices to think of a relation $f : A \leadsto B$ as a nondetermistic mapping between the source type $A$ and target type $B$. For example, the relation $split :: List\ A \leadsto (List\ A \times List\ A)$, which splits a list into two sublists, can be defined by

$$split\ (xs +\!\!+\ ys) \quad = \quad (xs, ys)$$

We write $y \leftarrow f\ x$ to denote that $y$ is a possible value to which $x$ is mapped through relation $f$. Take *split* defined above for example, both $([1],[2,3]) \leftarrow split\ [1,2,3]$ and $([1,2],[3]) \leftarrow split\ [1,2,3]$ hold.

For a relation $f$ we can find its domain and range:

$$\begin{array}{lcl} dom\ f & = & \{x \mid \exists y : y \in B : y \leftarrow f\ x\} \\ ran\ f & = & \{y \mid \exists x : x \in A : y \leftarrow f\ x\} \end{array}$$

Inclusion defines an ordering between relations: $f \subseteq g$ holds if and only if $(y \leftarrow f\ x) \Rightarrow (y \leftarrow g\ x)$. It is the model we choose for program refinement. An specification $expr_1$ can be refined to $expr_2$ if $expr_2 \subseteq expr_1$, while its domain is still preserved ($dom\ expr_1 \subseteq dom\ expr_2$).

Composition of two relations $f : B \leadsto C$ and $g : A \leadsto B$ is defined by $z \leftarrow (f \cdot g)\, x \equiv \exists y : y \leftarrow g\, x \wedge z \leftarrow f\, y$. The *converse* of a relation $f : A \leadsto B$, written $f^\circ : B \leadsto A$, is defined by $x \leftarrow f^\circ\, y \equiv y \leftarrow f\, x$. The converse is contravariant, that is, $(f \cdot g)^\circ = g^\circ \cdot f^\circ$.

Throughout this paper, we adopt the convention that $f :: A \rightarrow B$ denotes a (total) function from $A$ to $B$, while $f :: A \leadsto B$ denotes a relation.

Let $f$ be a relation of type $A \leadsto B$. For our purpose it suffices to say that the power transpose operator $\Lambda$ creates a function $\Lambda f :: A \rightarrow Set\, B$. For $a \in A$, $(\Lambda f)a$ is the set of all values in $B$ which $a$ is mapped to.

$$(\Lambda f)a \quad = \quad \{b | b \leftarrow fa\}$$

## 2.2 Inverting a function

The problem we are dealing with concerns building trees. Functional programmers are aware that flattening a structure is usually performed by a fold operation. Consequently, building a structure is usually performed by the converse operation of unfolding. However, this is not necessarily so. The converse of a function theorem tells us how we can write the inverse of a function as a fold. In this paper we will focus our attention on inverting functions whose range type is a non-empty list, therefore we only need a specialisation of the theorem to non-empty lists. In this paper we denote the type of non-empty lists of $A$s by $List^+\, A$.

**Theorem 1 (Converse of a Function Theorem (for non-empty lists))** *Let $f :: B \rightarrow List^+\, A$ be given. If $base :: A \rightarrow B$ and $step :: A \rightarrow B \leadsto B$ are jointly surjective ($ran\ base \cup ran\ step = B$) and satisfy*

$$f\,(base\ a) \quad = \quad [a] \tag{1}$$

$$f(step\ a\ x) \quad = \quad a : f\ x \tag{2}$$

*then $f^\circ = foldr\ step\ base$.*

## 2.3 Minimum

In $min\,(\trianglelefteq)$, the symbol $(\trianglelefteq)$ is an *ordering*. An ordering is

- *reflexive*, if $x \trianglelefteq x$ for all x;
- *anti-symmetric*, if $x \trianglelefteq y$ and $y \trianglelefteq x$ implies $x = y$;
- *transitive*, if $x \trianglelefteq y$ and $y \trianglelefteq z$ implies $x \trianglelefteq z$.

An ordering satisfying all the three conditions above is called a *partial order*. If it is reflexive and transitive but not necessarily anti-symmetric, it is called a *preorder*. That means we allow two different solutions to be equally preferred.

A preorder is connected if for all $x$ and $y$ of the correct type, either $x \trianglelefteq y$ or $y \trianglelefteq x$. That is, every two items can be compared. For a connected preorder $(\trianglelefteq)$, the relation $min\,(\trianglelefteq) :: Set\,A \rightsquigarrow A$ is defined by

$$x \leftarrow min\,(\trianglelefteq)\,xs \equiv x \in xs \wedge (\forall y : y \in xs : y \trianglelefteq x)$$

It is a relation rather than a function because $(\trianglelefteq)$ is not necessary anti-symmetric, which means that there may be more than one minimal element in a given set.

Among the many properties of $min$, we will in particular make use of the one below in the following sections. For a function $f$, (3) says that if we perform relation $f$ on every item in a set before selecting the minimal one under ordering $(\trianglelefteq)$, we can also do the selection in the range of $f$, then perform $f$ afterwards on the selected item only.

$$\begin{array}{l} f \cdot min\,(\preceq) \cdot \Lambda g \subseteq min\,(\trianglelefteq) \cdot \Lambda(f \cdot g) \\ \qquad \textbf{where } x \preceq y \equiv (f\ x) \trianglelefteq (f\ y) \end{array} \tag{3}$$

### 2.4 The greedy theorem

The greedy theorem plays the key role in the derivation in this paper.

**Theorem 2 (Greedy Theorem (for non-empty lists))** *Let $base :: A \rightsquigarrow B$ and $step :: A \to B \rightsquigarrow B$ be relations. If step is monotonic on connected preorder $(\trianglelefteq)$, on the sense that*

$$y \trianglelefteq x \wedge x' \leftarrow step\ a\ x \quad \Rightarrow \quad (\exists y' : y' \leftarrow step\ a\ y : y' \trianglelefteq x') \tag{4}$$

*then*

$$foldrn\,(min\,(\trianglelefteq) \cdot \Lambda step)\,(min\,(\trianglelefteq) \cdot \Lambda base) \subseteq min\,(\trianglelefteq) \cdot \Lambda(foldrn\ step\ base)$$

The expression *foldrn step base* on the right-hand side generates all the items to be compared by $min\,(\trianglelefteq)$ by folding over the input list. The relation *base* gives us some items to start with, while *step* takes an item and extends it. The monotonicity condition above in effect means that for two items $x$ and $y$, $y$ being at least as good as $x$ (with respect to $(\trianglelefteq)$), no matter how we extend $x$ to $x'$, we can always find a way to extend $y$ to $y'$ such that $y'$ is not worse than $x'$. There is thus no point keeping the worse one, $x$, in the first place. We need only to keep the *best* item so far in each stage.

Therefore, we can promote $min\,(\trianglelefteq)$ into *foldrn*. Rather than looking for the minimal one among all the items returned by *foldrn*, the minimal one is chosen in each step and is the only one passed to the next step.

## 3 The derivation

### 3.1 Problem definition

Consider the following datatype definition for tip-valued binary trees

$$\textbf{data}\ Tree\ A \quad = \quad Tip\ A \mid Bin\,(Tree\ A)\,(Tree\ A)$$

The function computing the height of a tree and flattening a tree can be defined as folds over *Tree A* in the obvious way:

$$
\begin{array}{lcl}
height & :: & Tree\ A \to A \\
height & = & foldTree\ (\oplus)\ id \\
 & & \mathbf{where}\ a \oplus b = max\ a\ b + 1
\end{array}
$$

$$
\begin{array}{lcl}
flatten & :: & Tree\ A \to List^{+}\ A \\
flatten & = & foldTree\ (+\!\!+)\ wrap
\end{array}
$$

where $wrap\ a = [a]$ wraps its argument into a singleton list and *foldTree* is the fold function for *Tree* defined by:

$$
\begin{array}{lcl}
foldTree\ f\ g\ (Tip\ a) & = & g\ a \\
foldTree\ f\ g\ (Bin\ x\ y) & = & f\ (foldTree\ f\ g\ x)\ (foldTree\ f\ g\ y)
\end{array}
$$

The problem is thus to find, among all the trees which flattens to the given list, the one(s) for which *height* yields the minimal value.

The inverse of the function *flatten*, written $flatten^{\circ}$, relates a list to any tree which flattens to it. It must be a relation because there are in general many trees which could flatten to the same list. The $\Lambda$ operator enables us to talk about the set of all such trees as a whole. The expression $\Lambda(flatten^{\circ})$, having type $List^{+}\ A \to Set(Tree\ A)$, returns the set of all the trees which flatten to the given list. For the purpose of our problem, we choose

$$
x \trianglelefteq y \quad \equiv \quad height\ x \leq height\ y
$$

The problem can then be specified as

$$
build \quad = \quad min\ (\trianglelefteq) \cdot \Lambda(flatten^{\circ})
$$

## 3.2 Building the tree

As *flatten* is a fold, $flatten^{\circ}$ can be a unfold. There is indeed a greedy theorem for problems specified in terms of unfold. In this paper, however, we will make use of the converse of the function theorem and write $flatten^{\circ}$ as a fold.

An alternative way of representing a tree is the *spine representation*, in which a tree is represented by the list of subtrees along the left spine, plus the left-most tip. The function *roll* converts a spine into the ordinary representation, with the help of the Prelude function *foldl*. It is in fact an isomorphism between *Spine A* and *Tree A*.

$$
\begin{array}{lcl}
\mathbf{type}\ Spine\ A & = & (A \times List(Tree\ A))
\end{array}
$$

$$
\begin{array}{lcl}
roll & :: & Spine\ A \to Tree\ A \\
roll(a, x) & = & foldl\ Bin\ (Tip\ a)\ x
\end{array}
$$

The task is to find an inverse for $flatten \cdot roll$. According to the converse of a function theorem, we need a pair of functions *one* and *add* that are jointly surjective and satisfy

$$
\begin{array}{lcl}
f(one\ a) & = & [a] \\
f(add\ a\ x) & = & a : f\ x
\end{array}
$$

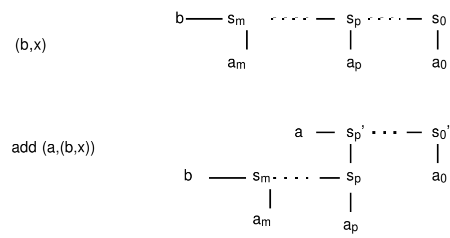


Figure 2: An example of *add* breaking spine $(b, x)$ at position $p$, where $x = y \mathbin{+\!\!+} z$, $y = [a_m, a_{m-1}, \ldots a_p]$ and $z = [a_{p-1}, a_{p-2}, \ldots a_0]$. The $s_i$'s are the values on the spine. They are not actually represented in our data structure. After extending $x$, spine values $s_p, s_{p-1}, \ldots s_0$ are updated.

We claim that the following definition for *one* and *add* satisfies the premises.

$$\begin{array}{lcl} \mathit{one}\ a & = & (a, [\,]) \\ \mathit{add}\ a\ (b, x) & = & (a, \mathit{roll}\ (b, y) : z) \\ & & \quad \textbf{where}\ y \mathbin{+\!\!+} z = x \end{array}$$

The condition for *one* trivially holds. In the definition of *add*, we use a nondetermistic pattern on the left hand side to break the list $x$ into two parts. Such a matching is always possible because $(\mathbin{+\!\!+})$ is surjective. To show that *add* meets the condition, we will need the following fact:

$$\mathit{flatten}(\mathit{roll}\ (a, x)) \quad = \quad a : \mathit{concat}(\mathit{map}\ \mathit{flatten}\ x) \tag{5}$$

We reason

$$\begin{array}{cl} & a : \mathit{flatten}(\mathit{roll}\ (b, y \mathbin{+\!\!+} z)) \\ = & \quad \{(5)\} \\ & a : b : \mathit{concat}(\mathit{map}\ \mathit{flatten}\ (y \mathbin{+\!\!+} z)) \\ = & \quad \{\mathit{concat} \text{ and } \mathit{map} \text{ distributes over } \mathbin{+\!\!+}\ \} \\ & a : b : \mathit{concat}(\mathit{map}\ \mathit{flatten}\ y) \mathbin{+\!\!+} \mathit{concat}(\mathit{map}\ \mathit{flatten}\ z) \\ = & \quad \{(5), \text{backwards}\} \\ & a : \mathit{flatten}(\mathit{roll}\ (b, y)) \mathbin{+\!\!+} \mathit{concat}(\mathit{map}\ \mathit{flatten}\ z) \\ = & \quad \{\text{definition of } \mathit{concat} \text{ and } \mathit{map}\} \\ & a : \mathit{concat}(\mathit{map}\ \mathit{flatten}\ (\mathit{roll}(b, y) : z)) \\ = & \quad \{(5), \text{backwards}\} \\ & \mathit{flatten}\ (\mathit{roll}\ (a, \mathit{roll}(b, y) : z)) \\ = & \quad \{\text{definition of } \mathit{add}\} \\ & \mathit{flatten}\ (\mathit{roll}\ (\mathit{add}\ a\ (b, y \mathbin{+\!\!+} z))) \end{array}$$

Thus we have $(\mathit{flatten} \cdot \mathit{roll})^\circ = \mathit{foldrn}\ \mathit{add}\ \mathit{one}$. Intuitively, we build the spine tree by folding over the non-empty list of values, inserting nodes into the tree one by one. Relation *add* breaks the spine $(b, x)$ in an arbitrary position and attaches $a$ to the end, as shown in Fig. 2. There are many ways to break $x$ into $y \mathbin{+\!\!+} z$, which is where the nondeterminism comes from. We will see later that eliminating or reducing this nondeterminism is the key toward deriving an efficient algorithm for optimal bracketing problem.

Having inverted $\mathit{flatten} \cdot \mathit{roll}$, we can now rephrase our problem definition:

$$
\begin{array}{cl}
 & \mathit{build} \\
= & \quad \{\text{specification}\} \\
 & \mathit{min}\,(\trianglelefteq) \cdot \Lambda(\mathit{flatten}^{\circ}) \\
= & \quad \{\mathit{roll} \text{ isomorphic}\} \\
 & \mathit{min}\,(\trianglelefteq) \cdot \Lambda((\mathit{flatten} \cdot \mathit{roll} \cdot \mathit{roll}^{\circ})^{\circ}) \\
= & \quad \{\text{converse contravariant}\} \\
 & \mathit{min}\,(\trianglelefteq) \cdot \Lambda(\mathit{roll} \cdot (\mathit{flatten} \cdot \mathit{roll})^{\circ}) \\
= & \quad \{\mathit{roll} \text{ function, (3), define } x \preceq y \equiv (\mathit{roll}\; x) \trianglelefteq (\mathit{roll}\; y)\} \\
 & \mathit{roll} \cdot \mathit{min}\,(\preceq) \cdot \Lambda(\mathit{flatten} \cdot \mathit{roll})^{\circ} \\
= & \quad \{\text{inverting } \mathit{flatten} \cdot \mathit{roll}\} \\
 & \mathit{roll} \cdot \mathit{min}\,(\preceq) \cdot \Lambda(\mathit{foldrn}\; \mathit{add}\; \mathit{one})
\end{array}
$$

The specification naively generates an exponential number of all possible trees flattening to the same list, and choose an optimal one with respect to preorder $(\preceq)$.

### 3.3 A greedy algorithm

If we can prove that *add* satisfies the monotonicity condition (4), then we can have a greedy algorithm. However, it is not true with respect to $(\preceq)$: a tree with the smallest height does not always remain the best after being extended by *add*.

Fortunately, *add* is monotonic on a stronger ordering. If we take $(\ll)$ to be the reversed *lexicographic ordering* on the values along the left spine (that is, first the values on the roots are compared, if they are equal, then the next values on the spines are compared.. and so on), we can prove the monotonic condition with respect to $(\ll)$. This choice does make sense: to ensure monotonicity, we need to optimise not only the whole tree, but also all the subtrees on the left spine.

We will explain how we can maintain the monotonicity. For any two spine trees $x \ll y$, no matter how $x$ is extended by *add*, we must find a way to extend $y$ such that the resulting tree is not worse. Suppose the spines of $x$ and $y$ look like in Fig. 3. Note that values on each spines are strictly increasing. Furthermore, by bringing in the context, we can assume that $s_m = t_n$. Therefore,

1. either the spine values are all the same (i.e. $m = n \;\wedge\; \forall i : m \geq i \geq 0 : t_i = s_i$), or

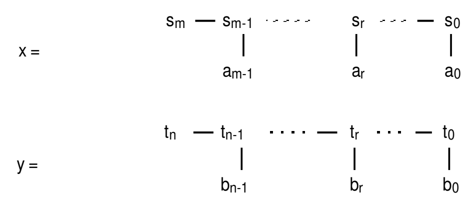

Figure 3: Assumption of how $x$ and $y$ look like. Here $x = (s_m, [a_{m-1}, a_{m-2}, \ldots a_0])$ and $y = (t_n, [b_{n-1}, b_{n-2}, \ldots b_0])$. $s_{m-1}, s_{m-2}, \ldots s_0$ and $t_{n-1}, t_{n-2}, \ldots t_0$ are the values on the spine.

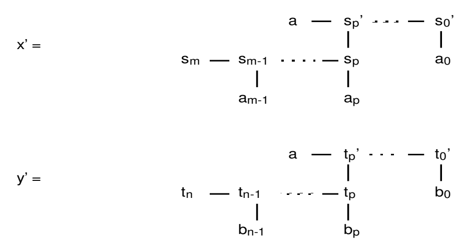

Figure 4: How we can extend $y$ when $p < r$.

2. we can find the first value on $x$, starting from the root, strictly greater than the corresponding value on $y$. That is, exists $r$ between $m \sqcap n$ and 0 such that $s_r > t_r$ and $\forall i : r > i \geq 0 : t_i = s_i$.

Assume $x$ was extended at position $p$. If the position comes before the two spines divert from each other, that is, $0 \leq p \leq r$, we always extend $y$ at the same position $p$, as in Fig. 4. If $p$ comes after $r$, that is, $p > r$, we can always extend $y$ at position $r$, as in Fig. 5.

Once we know the monotonicity condition holds, we can apply the greedy theorem to refine the specification to

$$roll \cdot foldrn\,(min(\ll) \cdot \Lambda add)\,(min(\ll) \cdot \Lambda one)$$

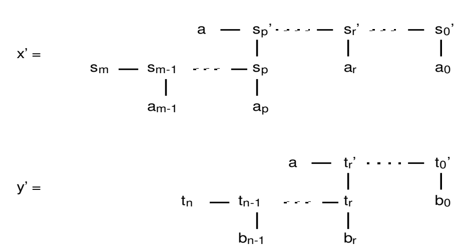

Figure 5: How we can extend $y$ when $p \geq r$.

```
data Tree a = Tip a | Bin (Tree a) (Tree a)
type SpineC a = (a, [(a,Tree a)])

build :: [Integer] -> Tree Integer
build = rollC . foldrn minadd one

rollC :: SpineC a -> Tree a
rollC (a,x) = foldl join (Tip a) x
  where join x (a,y) = Bin x y

one a = (a,[])

minadd :: Integer -> SpineC Integer -> SpineC Integer
minadd a (b,xs) = (a, minsplit ((b,Tip b):xs))
  where minsplit [x] = [x]
        minsplit (x:y:xs) | a < fst y && fst x < fst y = x:y:xs
                          | otherwise = minsplit (bin x y:xs)
        bin (a,x) (b,y) = (ht a b, Bin x y)

ht a b = (a `max` b) + 1
```

Figure 6: Program for Building Trees with Minimum Height

Since *one* is a function, it is equivalent to

$$roll \cdot foldrn\,(min(\ll) \cdot \Lambda add)\; one$$

## 3.4 A further refinement and the code

Further improvements can be made during refining $min\,(\ll) \cdot \Lambda add$. We can prove that to find the best position to insert node $a$ on spine $x$ of length $n$, we do not need to actually check through all the $n+1$ possibilities. The minimal result always came from extending $x$ at position $p$, where $p$ is the maximal index satisfying $a \oplus s_p < s_{p-1}$, assuming $s_{-1} = \infty$. We can start from the left of the spine and choose the first $p$ which satisfies the condition.

In the implementation, we refine the data structure to avoid recomputing the height of each subtree. A spine is represented by **type** $SpineC\ a = (A, List(A \times Tree\ A))$, annotating each subtree along the spine with its height.

The resulting code is shown in Fig. 6, where function `build` takes a list of integers, denoting tree heights, and returns the tree built out of the list whose height is minimum. Function `minadd` is the result of the refinement described in this section. It's not difficult to see that it is a linear time algorithm, since each call to `minadd` consumes a value, each recursive call to `minsplit` either returns or joins a node, and each node in the resulting tree is built only once.

## 4 Conclusion and related work

We have demostrated how the old problem of building trees of minimum height can be rephrased in the new scenario of optimisation problems. To put the problem in the right form, we made use of the converse of a function theorem to express the construction of a tree as a fold on lists. The greedy theorem can then be applied.

At least two reasons justify the generalisation from function to relations. Firstly, it provides us a clean way to express nondeterminism. It is especially convenient when we consider optimisation problem, for which we have multiple choices to make in each step. Without relations, we would have to keep the solutions in lists and mess the specification with bookkeeping details taking care of the lists. Secondly, the inverse of a function is not in general a function. We need to generalise to relations before we can talk about it at all.

The motivation behind this line of research is to extend the result to the optimal bracketing problem in general, for which a non-connected preorder is used and the *thinning theorem*, a generalisation of the greedy theorem, is applied. The relationship between the thinning theorem and the traditional dynamic programming approach remains a topic for further research.

The greedy theorem, together with a family of theorems useful for optimisation problems has been studied extensively in [2]. The converse of a function theorem and the idea of the spine representation was adopted from [3]. The former has proved to be useful in many applications. The latter, however, is relatively less known. Work is still in progress to find more applications of the converse of a function theorem, as many problems can be specified in terms of the inverse of some known function. The general form of all the three theorems for arbitrary inital datatypes and their proofs can be found in [2].

## References

[1] R. Bird. On Building Trees with Minimum Height. *Journal of Functional Programming*, 7(4):441–445, 1997.

[2] R. Bird and O. de Moor. *Algebra of Programming*. International Series in Computer Science. Prentice Hall, 1997.

[3] O. de Moor and J. Gibbons. Pointwise Relational Programming. In *Algebraic Methodology and Software Technology*, May 2000.