Calculating a functional module for binary search trees

Walter Dosch, Bernhard Möller

Angaben zur Veröffentlichung / Publication details:

Dosch, Walter, and Bernhard Möller. 1997. "Calculating a functional module for binary search trees." *Lecture Notes in Computer Science* 1268: 267–84. https://doi.org/10.1007/3-540-63237-9_30.

Calculating a Functional Module for Binary Search Trees

Walter Dosch[1] and Bernhard Möller[2]

[1] Institut für Softwaretechnik und Programmiersprachen,
Medizinische Universität zu Lübeck, D–23538 Lübeck
[2] Institut für Informatik, Universität Augsburg, D–86135 Augsburg

Abstract. We formally derive a functional module for binary search trees comprising search, insert, delete, minimum and maximum operations. The derivation starts from an extensional specification that refers only to the multiset of elements stored in the tree. The search tree property is systematically derived as an implementation requirement.

1 Introduction

Search trees are a well-known dynamic data structure for storing and retrieving data [AHU83, CLR90, K73, Mh84, S88]. There are various types of search trees used in different applications. The algorithms for binary search trees are not complex; nevertheless none of the standard text books on data structures verifies their correctness rigorously or — more desirably — calculates them from the specification.

In the present paper we systematically derive a functional module for binary search trees comprising search, insertion, deletion, minimum, and maximum operations. The binary tree represents a multiset, since there may be multiple occurrences of the same element. The functions on search trees are specified by referring only to this multiset, but not to the internal structure of trees.

In the derivation we stress algebraic calculation rather than logical deduction. The development is carried out at the level of functional programs; it widely exploits the algebraic properties of the underlying data structures.

The formal derivation of the algorithms gives insight into the algorithmic principles underlying search trees. In particular, we precisely locate the simplifications originating from the *binary search tree property* stating that each node is an upper (lower) bound for the nodes in the left (right) subtree. This constraint is not imposed from the beginning; rather it is derived as an implementation requirement. We present functional techniques for the joint development of the data and the control structure.

Throughout the paper we show how the creative steps in the development, viz. abstracting and generalizing subtasks, naturally arise from simple calculations using induction over the tree structure. We base the derivation on weak assumptions; this avoids overspecification and preserves the freedom for further refinements.

A functional algorithm for inserting elements into 2-3-trees was already given in [HD83]. For the deletion of elements in 2-3 trees, a rigorous correctness proof involving subtypes was provided by [R92]. In contrast to the backward oriented verification of a given algorithm, this paper concentrates on deductive design from the specification in forward direction.

We assume that the reader has a basic knowledge of functional programming (for overviews see [B89, H89]) and of transformational program design (see [BMPP89, Me86, P90]). Throughout the paper we use traditional mathematical notation. Functions are defined using axioms

$$t = \begin{cases} t_1 & \text{if } \alpha \\ t_2 & \text{if } \beta \end{cases}$$

with terms t, t_1, t_2 . Such an axiom is by definition equivalent to

$$\begin{array}{l} ((\alpha \wedge \neg\beta) \Rightarrow t = t_1) \wedge \\ ((\neg\alpha \wedge \beta) \Rightarrow t = t_2) \wedge \\ \quad ((\alpha \wedge \beta) \Rightarrow (t = t_1 \vee t = t_2)) \,. \end{array}$$

If both conditions α and β hold, the overlapping patterns do not lead to a contradiction; rather they express an underspecification, since then the formula is satisfied by a *set of functions*. It is a matter of later refinement to determine which of these functions actually is taken as the implementation.

2 The Data Structures

Let $(\mathcal{B}; \wedge, \vee, \neg)$ be the Boolean algebra of *truth values* with $\mathcal{B} = \{T, F\}$. Furthermore let $(\mathcal{E}; \leq, \mathit{min}, \mathit{max}, -\infty, +\infty)$ be the set of *elements*; it is equipped with a linear order possessing smallest and greatest elements.

2.1 Multisets

The specification of search trees refers to the multiset of data stored. For simplicity, we take as data just (atomic) elements.

The Multiset Algebra The algebra of multisets with elements from $\mathcal{E}$ comprises the set $\mathcal{M}$ of all *finite multisets* inductively generated by the following constant and operations:

$$\begin{array}{rr} \emptyset \in \mathcal{M} & \{\text{empty multiset}\} \\ \{.\} : \mathcal{E} \to \mathcal{M} & \{\text{forming singleton multisets}\} \\ .\cup. : \mathcal{M} \times \mathcal{M} \to \mathcal{M} & \{\text{multiset union}\} \end{array}$$

The algebra $(\mathcal{M}, .\cup., \emptyset)$ forms a commutative monoid:

$$(m \cup n) \cup o = m \cup (n \cup o) \tag{1}$$

$$\emptyset \cup m = m = m \cup \emptyset \tag{2}$$

$$m \cup n = n \cup m \tag{3}$$

The containment relation $.\in.: \mathcal{E} \times \mathcal{M} \to \mathcal{B}$ satisfies the equations

$$x \in \emptyset = F \tag{4}$$
$$x \in \{y\} = (x = y) \tag{5}$$
$$x \in (m \cup n) = x \in m \vee x \in n\,. \tag{6}$$

The deletion $.\backslash.: \mathcal{M} \times \mathcal{E} \to \mathcal{M}$ of an element obeys the laws

$$\emptyset\backslash y = \emptyset \tag{7}$$
$$\{x\}\backslash y = \begin{cases} \emptyset & \text{if } y = x \\ \{x\} & \text{if } y \neq x \end{cases} \tag{8}$$
$$(m \cup n)\backslash y = (m\backslash y) \cup n \qquad \text{if } y \in m \vee y \notin n \tag{9}$$

The premise of equation (9) can always be established using the commutativity of multiset union (3), for example

$$(\{y\} \cup \{x\})\backslash x = (\{x\} \cup \{y\})\backslash x = (\{x\}\backslash x) \cup \{y\} = \emptyset \cup \{y\} = \{y\}\,.$$

The smallest element $minel\colon \mathcal{M} \to \mathcal{E}$ of a multiset is defined by the equations

$$minel(\emptyset) = +\infty \tag{10}$$
$$minel(\{x\}) = x \tag{11}$$
$$minel(m \cup n) = min(minel(m), minel(n))\,; \tag{12}$$

the greatest element $maxel\colon \mathcal{M} \to \mathcal{E}$ is defined symmetrically. This completes the definition of the multiset algebra $\mathcal{M}$.

Lower and Upper Bounds The linear order $\leq$ on the set $\mathcal{E}$ of elements induces a lower bound relation between elements and multisets:

$$x \sqsubseteq \emptyset = T \tag{13}$$
$$x \sqsubseteq \{y\} = x \leq y \tag{14}$$
$$x \sqsubseteq m \cup n = x \sqsubseteq m \wedge x \sqsubseteq n \tag{15}$$

The upper bound relation is defined symmetrically:

$$\emptyset \sqsubseteq y = T \tag{16}$$
$$\{x\} \sqsubseteq y = x \leq y \tag{17}$$
$$m \cup n \sqsubseteq y = m \sqsubseteq y \wedge n \sqsubseteq y \tag{18}$$

The corresponding strict lower and strict upper bound relations are defined analogously. The bound relations are transitive in the following sense:

$$y < x \wedge x \sqsubseteq m \;\Rightarrow\; y \sqsubset m \tag{19}$$
$$m \sqsubseteq x \wedge x < y \;\Rightarrow\; m \sqsubset y \tag{20}$$

Strict bound relations allow concluding non-membership:

$$x \sqsubset m \;\Rightarrow\; x \notin m \tag{21}$$
$$m \sqsubset x \;\Rightarrow\; x \notin m \tag{22}$$

Moreover, bound relations are monotonic wrt. the deletion of elements:

$$m \sqsubseteq x \;\Rightarrow\; m\backslash y \sqsubseteq x \tag{23}$$
$$x \sqsubseteq m \;\Rightarrow\; x \sqsubseteq m\backslash y \tag{24}$$

The computation of the smallest and greatest elements of a multiset union can be simplified if one of the multisets is a singleton $\{x\}$ and the other is bounded by x:

$$x \sqsubseteq m \Rightarrow \mathit{minel}(\{x\} \cup m) = x \tag{25}$$
$$x \sqsubseteq m \Rightarrow \mathit{maxel}(\{x\} \cup m) = \begin{cases} x & \text{if } m = \emptyset \\ \mathit{maxel}(m) & \text{if } m \neq \emptyset \end{cases} \tag{26}$$

Symmetrically we have:

$$m \sqsubseteq x \Rightarrow \mathit{maxel}(m \cup \{x\}) = x \tag{27}$$
$$m \sqsubseteq x \Rightarrow \mathit{minel}(m \cup \{x\}) = \begin{cases} x & \text{if } m = \emptyset \\ \mathit{minel}(m) & \text{if } m \neq \emptyset \end{cases} \tag{28}$$

Finally, the extremal elements of non-empty multisets are related to bounds. For $m \neq \emptyset$ we have:

$$x = \mathit{minel}(m) \;\Leftrightarrow\; x \in m \wedge x \sqsubseteq m \tag{29}$$
$$x = \mathit{maxel}(m) \;\Leftrightarrow\; x \in m \wedge m \sqsubseteq x \tag{30}$$

Relations (19)–(30) can all be shown by simple structural induction on multisets.

2.2 Binary Trees

The set $\mathcal{T}$ of *binary trees* with elements $\mathcal{E}$ as nodes is defined inductively as the least set with

$$(i)\;\; \varepsilon \in \mathcal{T} \tag{31}$$
$$(ii)\;\; \mathcal{T} \times \mathcal{E} \times \mathcal{T} \subseteq \mathcal{T}\,, \tag{32}$$

where $\times$ is the ternary, non-associative cartesian product. In the sequel, ε denotes the empty binary tree while the triple $\langle l, x, r\rangle$ denotes a non-empty binary tree with left subtree $l \in \mathcal{T}$, node $x \in \mathcal{E}$, and right subtree $r \in \mathcal{T}$.

2.3 Representation

The multiset of elements a binary tree represents is obtained by forgetting the tree structure. The corresponding abstraction function $multi$: $\mathcal{T} \to \mathcal{M}$ reads:

$$multi(\varepsilon) = \emptyset \tag{33}$$

$$multi(\langle l, x, r\rangle) = multi(l) \cup \{x\} \cup multi(r) \tag{34}$$

It is the unique homomorphism from the algebra $(\mathcal{T}; \varepsilon, \langle ., ., .\rangle)$ of binary trees to the algebra $(\mathcal{M}; \emptyset, . \cup \{.\} \cup .)$ of multisets. The empty and all singleton multisets are uniquely represented by trees:

$$multi(t) = \emptyset \text{ iff } t = \varepsilon \tag{35}$$

$$multi(t) = \{x\} \text{ iff } t = \langle \varepsilon, x, \varepsilon\rangle \tag{36}$$

The converse of the abstraction function $multi$ yields a one-to-many representation relation.

2.4 Derivation Techniques

Many of our functions f on search trees are specified implicitly in the form $multi(f(t)) = E$, where E is some expression in t not involving f. Our strategy then is to find some tree t' with $multi(t') = E$ as well. Then $f(t) = t'$ is a correct implementation. To avoid excessive tree rearrangements, it is advantageous to choose t' as similar to t as possible.

3 Search

The search function for an element in a binary tree is specified by referring to the associated multiset:

$$search\colon \mathcal{T} \times \mathcal{E} \to \mathcal{B}$$

$$search(t, y) = y \in multi(t) \tag{37}$$

3.1 Direct Recursion

We derive a direct recursion for the function $search$ (37) by induction on the tree structure (31)–(32) exploiting the laws of the containment relation. The comments on the right-hand side justify the deduction step between the formulas in the current and the subsequent line.

Induction basis

$$\begin{aligned} & search(\varepsilon, y) && \{\text{unfold } search\ (37)\} \\ = {} & y \in multi(\varepsilon) && \{\text{unfold } multi\ (33)\} \\ = {} & y \in \emptyset && \{\text{by } (4)\} \\ = {} & F \end{aligned}$$

Induction step

$$
\begin{aligned}
& \mathit{search}(\langle l, x, r\rangle, y) && \{\text{unfold } \mathit{search}\ (37)\}\\
= {} & y \in \mathit{multi}(\langle l, x, r\rangle) && \{\text{unfold } \mathit{multi}\ (34)\}\\
= {} & y \in \mathit{multi}(l) \cup \{x\} \cup \mathit{multi}(r) && \{\text{by } (6)\}\\
= {} & y \in \mathit{multi}(l) \vee y \in \{x\} \vee y \in \mathit{multi}(r) && \{\text{by } (5)\}\\
= {} & y \in \mathit{multi}(l) \vee (y = x) \vee y \in \mathit{multi}(r) && \{\text{fold } \mathit{search}\ (37)\}\\
= {} & \mathit{search}(l, y) \vee (y = x) \vee \mathit{search}(r, y) && \{\text{sequentialize disjunction}\}\\
= {} & \begin{cases} T & \text{if } y = x \\ \mathit{search}(l, y) \vee \mathit{search}(r, y) & \text{if } y \neq x \end{cases}
\end{aligned}
$$

With the last transformation step, we achieve early termination when finding an occurrence of the search element in the tree. In summary, the search can be implemented by the cascading recursion

$$\mathit{search}(\varepsilon, y) = F \tag{38}$$

$$\mathit{search}(\langle l, x, r\rangle, y) = \begin{cases} T & \text{if } y = x \\ \mathit{search}(l, y) \vee \mathit{search}(r, y) & \text{if } y \neq x \end{cases} \tag{39}$$

where both the left and the right subtrees of a composite tree are inspected.

3.2 Search Trees

The cascading recursion (39) of the function *search* simplifies to a linear recursion if one of the disjuncts reduces to the neutral element F of the disjunction. Hence we calculate a sufficient condition for the left disjunct:

$$
\begin{aligned}
& \mathit{search}(l, y) = F && \{\text{unfold } \mathit{search}\ (37)\}\\
= {} & y \notin \mathit{multi}(l) && \{\text{by (21) and (22)}\}\\
\overset{(\star)}{\Leftarrow} {} & y \sqsubset \mathit{multi}(l) \vee y \sqsupset \mathit{multi}(l) && \{\text{by (19) and (20)}\}\\
\overset{(\star\star)}{\Leftarrow} {} & y < x \sqsubseteq \mathit{multi}(l) \vee y > x \sqsupseteq \mathit{multi}(l)
\end{aligned}
$$

With the first design decision ($\star$) we implement the test for non-membership by the strict bound relations. In the next step ($\star\star$) we introduce a cut element to localize the test for the bound relations to the comparison of two elements. Symmetrically, we derive for the right disjunct

$$\mathit{search}(r, y) = F \;\Leftarrow\; y < x \sqsubseteq \mathit{multi}(r) \vee y > x \sqsupseteq \mathit{multi}(r)\,.$$

This allows refining the search function (39) as follows:

$$\mathit{search}(\langle l, x, r\rangle, y) = \begin{cases} T & \text{if } y = x \\ \mathit{search}(l, y) & \text{if } y < x \sqsubseteq \mathit{multi}(r) \vee y > x \sqsupseteq \mathit{multi}(r) \\ \mathit{search}(r, y) & \text{if } y < x \sqsubseteq \mathit{multi}(l) \vee y > x \sqsupseteq \mathit{multi}(l) \\ \mathit{search}(l, y) \vee \mathit{search}(r, y) & \text{otherwise}\,. \end{cases} \tag{40}$$

The disjuncts in condition (40) are mutually exclusive, since for a linear order $x \neq y$ implies either $y < x$ or $y > x$. Hence equation (40) allows two solutions,

either

$$search(\langle l, x, r\rangle, y) = \begin{cases} T & \text{if } y = x \\ search(l, y) & \text{if } y < x \sqsubseteq multi(r) \\ search(r, y) & \text{if } y > x \sqsupseteq multi(l) \\ search(l, y) \vee search(r, y) & \text{otherwise} \end{cases} \tag{41}$$

or the symmetrical version

$$search(\langle l, x, r\rangle, y) = \begin{cases} T & \text{if } y = x \\ search(l, y) & \text{if } y > x \sqsupseteq multi(r) \\ search(r, y) & \text{if } y < x \sqsubseteq multi(l) \\ search(l, y) \vee search(r, y) & \text{otherwise}\,. \end{cases} \tag{42}$$

Equation (41) simplifies to a linear recursion only if the bottom case "otherwise" never holds. To achieve this, we have to satisfy both conjuncts $x \sqsubseteq multi(r)$ and $x \sqsupseteq multi(l)$ as assertions in the data structure.

As the decisive design decision, we therefore confine the representation to binary trees where uniformly each node is an upper (lower) bound for the elements occurring in the left (right) subtree. This subset $\mathcal{S}$ of binary *search trees* is defined inductively by

$$(i)\ \ \varepsilon \in \mathcal{S} \tag{43}$$

$$(ii)\ \text{If } l, r \in \mathcal{S} \text{ and } x \in \mathcal{E} \text{ with } multi(l) \sqsubseteq x \sqsubseteq multi(r),\ \text{then } \langle l, x, r\rangle \in \mathcal{S}\,. \tag{44}$$

The set $\mathcal{S}$ of search trees is closed when forming subtrees, but not closed under the ternary tree constructor. The search function restricted to search trees

$$\begin{aligned} &\mathit{find}\colon\ \mathcal{S} \times \mathcal{E} \to \mathcal{B} \\ &\mathit{find} = search \mid (\mathcal{S} \times \mathcal{E}) \end{aligned} \tag{45}$$

then allows the intended simplification of (41) to

$$\mathit{find}(\varepsilon, y) = F \tag{46}$$

$$\mathit{find}(\langle l, x, r\rangle, y) = \begin{cases} \mathit{find}(l, y) & \text{if } y < x \\ T & \text{if } y = x \\ \mathit{find}(r, y) & \text{if } y > x\,. \end{cases} \tag{47}$$

The resulting search function *find* is tail-recursive and can be implemented by a simple loop.

In summary, the transformation to linear recursion exhibited the search tree property as an assertion on the data structure that allows replacing the blind (cascading) search by a strategic search based on the local comparison of elements.

4 Smallest and Greatest Elements

The smallest element of a search tree is specified by referring to the multiset representation:

$$\begin{aligned} &\mathit{minkey}\colon \mathcal{S} \to \mathcal{E} \\ &\mathit{minkey}(t) = \mathit{minel}(\mathit{multi}(t)) \end{aligned} \tag{48}$$

We derive a direct recursion by induction on the tree structure (43)–(44) exploiting the algebraic properties of least elements of multisets.

Induction basis

$$\begin{aligned} & \mathit{minkey}(\varepsilon) && \{\text{unfold } \mathit{minkey}\ (48)\} \\ = {} & \mathit{minel}(\mathit{multi}(\varepsilon)) && \{\text{unfold } \mathit{multi}\ (33)\} \\ = {} & \mathit{minel}(\emptyset) && \{\text{unfold } \mathit{minel}\ (10)\} \\ = {} & +\infty \end{aligned}$$

Induction step, assuming $\mathit{multi}(l) \sqsubseteq x \sqsubseteq \mathit{multi}(r)$.

$$\begin{aligned} & \mathit{minkey}(\langle l, x, r\rangle) && \{\text{unfold } \mathit{minkey}\ (48)\} \\ = {} & \mathit{minel}(\mathit{multi}(\langle l, x, r\rangle)) && \{\text{unfold } \mathit{multi}\ (34)\} \\ = {} & \mathit{minel}(\mathit{multi}(l) \cup \{x\} \cup \mathit{multi}(r)) && \{\text{by } (25)\} \\ = {} & \mathit{minel}(\mathit{multi}(l) \cup \{x\}) && \{\text{by } (28)\} \\ = {} & \begin{cases} x & \text{if } \mathit{multi}(l) = \emptyset \\ \mathit{minel}(\mathit{multi}(l)) & \text{if } \mathit{multi}(l) \neq \emptyset \end{cases} && \{\text{by } (35)\} \\ = {} & \begin{cases} x & \text{if } l = \varepsilon \\ \mathit{minkey}(l) & \text{if } l \neq \varepsilon \end{cases} && \{\text{fold } \mathit{minkey}\ (48)\} \end{aligned}$$

Again, the search tree property allows directing the search for the smallest element. In the resulting tail recursion

$$\mathit{minkey}(\varepsilon) = +\infty \tag{49}$$

$$\mathit{minkey}(\langle l, x, r\rangle) = \begin{cases} x & \text{if } l = \varepsilon \\ \mathit{minkey}(l) & \text{if } l \neq \varepsilon \end{cases} \tag{50}$$

the least element is found by successively visiting the left subtrees. The search function *maxkey* for the greatest element

$$\begin{aligned} &\mathit{maxkey}\colon \mathcal{S} \to \mathcal{E} \\ &\mathit{maxkey}(t) = \mathit{maxel}(\mathit{multi}(t)) \end{aligned} \tag{51}$$

can be derived symmetrically:

$$\mathit{maxkey}(\varepsilon) = -\infty \tag{52}$$

$$\mathit{maxkey}(\langle l, x, r\rangle) = \begin{cases} x & \text{if } r = \varepsilon \\ \mathit{maxkey}(r) & \text{if } r \neq \varepsilon \end{cases} \tag{53}$$

The functions *find*, *minkey* and *maxkey* derived so far only inspect the search tree, but do not modify it.

5 Insertion

The insertion of an element into a search tree is defined by the implicit specification

$$\begin{aligned} &\mathit{insert}\colon\ \mathcal{S} \times \mathcal{E} \to \mathcal{S} \\ &\mathit{multi}(\mathit{insert}(t, x)) = \mathit{multi}(t) \cup \{x\}\,. \end{aligned} \tag{54}$$

This specification leaves complete freedom how to restructure the search tree after the insertion of an element. We aim at deriving a direct recursion for the function *insert*. Since (54) makes sense for general trees in $\mathcal{T}$ as well, we first derive some necessary conditions by induction on the tree structure (31)–(32), exploiting properties of the multiset union.

Induction basis

$$\begin{aligned} & \mathit{multi}(\mathit{insert}(\varepsilon, y)) && \{\text{unfold } \mathit{insert}\ (54)\} \\ =\ & \mathit{multi}(\varepsilon) \cup \{y\} && \{\text{unfold } \mathit{multi}\ (33)\} \\ =\ & \emptyset \cup \{y\} && \{\text{by } (2)\} \\ =\ & \{y\} \end{aligned}$$

With (36) we conclude $\mathit{insert}(\varepsilon, y) = \langle \varepsilon, y, \varepsilon \rangle$.

Induction step, assuming $\langle l, x, r \rangle \in \mathcal{T}$.

$$\begin{aligned} & \mathit{multi}(\mathit{insert}(\langle l, x, r \rangle, y)) && \{\text{unfold } \mathit{insert}\ (54)\} \\ =\ & \mathit{multi}(\langle l, x, r \rangle) \cup \{y\} && \{\text{unfold } \mathit{multi}\ (34)\} \\ =\ & (\mathit{multi}(l) \cup \{x\} \cup \mathit{multi}(r)) \cup \{y\} && \{\text{by } (3)\} \\ =\ & \mathit{multi}(l) \cup \{x\} \cup (\mathit{multi}(r) \cup \{y\}) && \{\text{fold } \mathit{insert}\ (54)\} \\ =\ & \mathit{multi}(l) \cup \{x\} \cup \mathit{multi}(\mathit{insert}(r, y)) && \{\text{fold } \mathit{multi}\ (34)\} \\ =\ & \mathit{multi}(\langle l, x, \mathit{insert}(r, y) \rangle) \end{aligned}$$

Hence we may choose

$$\mathit{insert}(\langle l, x, r \rangle, y) = \langle l, x, \mathit{insert}(r, y) \rangle$$

in this case. Analogously we derive the symmetric equation and may choose

$$\mathit{insert}(\langle l, x, r \rangle, y) = \langle \mathit{insert}(l, y), x, r \rangle$$

as well. So for general trees the element could be inserted arbitrarily into the left or into the right subtree. For search trees we calculate the conditions under which insertion into the left subtree yields a search tree again.

Induction basis

$$\begin{aligned} & \langle \varepsilon, y, \varepsilon \rangle \in \mathcal{S} && \{\text{by } (44)\} \\ =\ & \varepsilon \in \mathcal{S} \wedge \mathit{multi}(\varepsilon) \sqsubseteq y \wedge y \sqsubseteq \mathit{multi}(\varepsilon) \wedge \varepsilon \in \mathcal{S} && \{\text{by } (43), \text{unfold } \mathit{multi}\ (33)\} \\ =\ & T \wedge \emptyset \sqsubseteq y \wedge y \sqsubseteq \emptyset \wedge T && \{\text{by } (13) \text{ and } (16)\} \\ =\ & T \end{aligned}$$

Induction step, assuming $\langle l, x, r\rangle \in \mathcal{S}$ and $\mathit{insert}(l, y) \in \mathcal{S}$.

$$
\begin{aligned}
&\langle \mathit{insert}(l, y), x, r\rangle \in \mathcal{S} && \{\text{by (44)}\}\\
&= \mathit{insert}(l, y) \in \mathcal{S} \wedge \mathit{multi}(\mathit{insert}(l, y)) \sqsubseteq x \wedge\ x \sqsubseteq \mathit{multi}(r) \wedge r \in \mathcal{S}\\
&&& \{\text{by assumption and induction hypothesis}\}\\
&= \mathit{multi}(\mathit{insert}(l, y)) \sqsubseteq x && \{\text{unfold } \mathit{insert}\ (54)\}\\
&= \mathit{multi}(l) \cup \{y\} \sqsubseteq x && \{\text{by (18)}\}\\
&= \mathit{multi}(l) \sqsubseteq x \wedge \{y\} \sqsubseteq x && \{\text{by assumption}\}\\
&= \{y\} \sqsubseteq x && \{\text{by (17)}\}\\
&= y \leq x
\end{aligned}
$$

Symmetrically, if $\langle l, x, r\rangle \in \mathcal{S}$ and $\mathit{insert}(r, y) \in \mathcal{S}$, then $\langle l, x, \mathit{insert}(r, y)\rangle \in \mathcal{S}$ simplifies to $y \geq x$. This establishes the soundness of the final solution

$$\mathit{insert}(\varepsilon, y) = \langle \varepsilon, y, \varepsilon\rangle \tag{55}$$

$$\mathit{insert}(\langle l, x, r\rangle, y) = \begin{cases} \langle \mathit{insert}(l, y), x, r\rangle & \text{if } y \leq x\\ \langle l, x, \mathit{insert}(r, y)\rangle & \text{if } y \geq x\,. \end{cases} \tag{56}$$

The conditions of equation (56) overlap for $x = y$: an element that occurs multiply can be added either to the left or to the right subtree. The equations (55)–(56) do not specify a single insertion function, rather a set of possible insertion functions; this leaves room for further design decisions. We can also narrow the choice uniformly for all nodes, for example, and strengthen the condition $y \geq x$ in (56) to $y > x$.

6 Deletion

The deletion of an element from a search tree is again specified implicitly by referring to the associated multiset:

$$
\begin{aligned}
&\mathit{delete}\colon \mathcal{S} \times \mathcal{E} \to \mathcal{S}\\
&\mathit{multi}(\mathit{delete}(t, y)) = \mathit{multi}(t) \backslash y
\end{aligned} \tag{57}
$$

6.1 Direct Recursion

We try to derive a direct recursion by induction on the structure of search trees (43)–(44) using the properties of the deletion operator on multisets.

Induction basis

$$
\begin{aligned}
&\mathit{multi}(\mathit{delete}(\varepsilon, y)) && \{\text{unfold } \mathit{delete}\ (57)\}\\
&= \mathit{multi}(\varepsilon) \backslash y && \{\text{unfold } \mathit{multi}\ (33)\}\\
&= \emptyset \backslash y && \{\text{by (7)}\}\\
&= \emptyset
\end{aligned}
$$

With (35) we conclude $\mathit{delete}(\varepsilon, y) = \varepsilon$.

Induction step, assuming $\mathit{multi}(l) \sqsubseteq x \sqsubseteq \mathit{multi}(r)$. To increase readability, we separate the cases.

Case $y < x$

$$
\begin{aligned}
& \mathit{multi}(\mathit{delete}(\langle l, x, r\rangle, y)) && \{\text{unfold } \mathit{delete}\ (57)\}\\
=\ & \mathit{multi}(\langle l, x, r\rangle)\backslash y && \{\text{unfold } \mathit{multi}\ (34)\}\\
=\ & (\mathit{multi}(l) \cup \{x\} \cup \mathit{multi}(r))\backslash y && \\
& && \{\text{by (9) since } y \notin \{x\} \cup \mathit{multi}(r) \text{ by (19) and (21)}\}\\
=\ & (\mathit{multi}(l)\backslash y) \cup \{x\} \cup \mathit{multi}(r) && \{\text{fold } \mathit{delete}\ (57)\}\\
=\ & \mathit{multi}(\mathit{delete}(l, y)) \cup \{x\} \cup \mathit{multi}(r) && \{\text{fold } \mathit{multi}\ (34)\}\\
=\ & \mathit{multi}(\langle \mathit{delete}(l, y), x, r\rangle) &&
\end{aligned}
$$

Case $y = x$

$$
\begin{aligned}
& \mathit{multi}(\mathit{delete}(\langle l, x, r\rangle, y)) && \{\text{unfold } \mathit{delete}\ (57)\}\\
=\ & \mathit{multi}(\langle l, x, r\rangle)\backslash y && \{\text{unfold } \mathit{multi}\ (34)\}\\
=\ & (\mathit{multi}(l) \cup \{x\} \cup \mathit{multi}(r))\backslash y && \{\text{by (9) since by (5) } y \in \{x\}\}\\
=\ & \mathit{multi}(l) \cup (\{x\}\backslash y) \cup \mathit{multi}(r) && \{\text{by (8) and design decision}\}\\
=\ & \mathit{multi}(l) \cup \emptyset \cup \mathit{multi}(r) && \{\text{by (2)}\}\\
=\ & \mathit{multi}(l) \cup \mathit{multi}(r) &&
\end{aligned}
$$

It is easily checked that the resulting partial solution

$$\mathit{delete}(\varepsilon, y) = \varepsilon \tag{58}$$

$$\mathit{delete}(\langle l, x, r\rangle, y) = \begin{cases} \langle \mathit{delete}(l, y), x, r\rangle & \text{if } y < x \\ \langle l, x, \mathit{delete}(r, y)\rangle & \text{if } y > x \end{cases} \tag{59}$$

fulfills the search tree condition:

Induction basis

$$
\begin{aligned}
& \mathit{delete}(\varepsilon, y) \in \mathcal{S} && \{\text{unfold } \mathit{delete}\ (58)\}\\
=\ & \varepsilon \in \mathcal{S} && \{\text{by (43)}\}\\
=\ & T &&
\end{aligned}
$$

Induction step for $y < x$, assuming $\langle l, x, r\rangle \in \mathcal{S}$ and $\mathit{delete}(l, y) \in \mathcal{S}$.

$$
\begin{aligned}
& \mathit{delete}(\langle l, x, r\rangle, y) \in \mathcal{S} && \{\text{unfold } \mathit{delete}\ (59)\}\\
=\ & \langle \mathit{delete}(l, y), x, r\rangle \in \mathcal{S} && \{\text{by (44)}\}\\
=\ & \mathit{delete}(l, y) \in \mathcal{S} \wedge \mathit{multi}(\mathit{delete}(l, y)) \sqsubseteq x \wedge x \sqsubseteq \mathit{multi}(r) \wedge r \in \mathcal{S} && \\
& && \{\text{by assumption and induction hypothesis}\}\\
=\ & \mathit{multi}(\mathit{delete}(l, y)) \sqsubseteq x && \{\text{unfold } \mathit{delete}\ (57)\}\\
=\ & \mathit{multi}(l)\backslash y \sqsubseteq x && \{\text{by (23)}\}\\
\Leftarrow\ & \mathit{multi}(l) \sqsubseteq x && \{\text{by assumption}\}\\
=\ & T &&
\end{aligned}
$$

The case $y > x$ is treated symmetrically. If the element to be deleted is found at the root of the search tree, we arrive at a new task, viz. deleting that root.

6.2 Root Deletion

The deletion of the root of a nonempty search tree is specified by

$$\mathit{delroot}\colon \mathcal{S}\backslash\{\varepsilon\} \to \mathcal{S}$$

$$\mathit{multi}(\mathit{delroot}(\langle l, x, r\rangle)) = \mathit{multi}(l) \cup \mathit{multi}(r)\,. \tag{60}$$

We aim at deriving a direct recursion, for example by induction on the structure (43)–(44) of the search tree l.

Induction basis

$$\begin{aligned}
& \mathit{multi}(\mathit{delroot}(\langle \varepsilon, x, r\rangle)) && \{\text{unfold } \mathit{delroot}\ (60)\}\\
&= \mathit{multi}(\varepsilon) \cup \mathit{multi}(r) && \{\text{unfold } \mathit{multi}\ (33)\}\\
&= \emptyset \cup \mathit{multi}(r) && \{\text{by } (2)\}\\
&= \mathit{multi}(r)
\end{aligned}$$

Induction step, assuming $l \neq \varepsilon$.

$$\begin{aligned}
& \mathit{multi}(\mathit{delroot}(\langle l, x, r\rangle)) && \{\text{unfold } \mathit{delroot}\ (60)\}\\
&= \mathit{multi}(l) \cup \mathit{multi}(r) && \{\text{split } \mathit{multi}(l) \neq \emptyset \text{ by introducing } \mathit{some}\}\\
&= (\mathit{multi}(l) \backslash \mathit{some}(l)) \cup \{\mathit{some}(l)\} \cup \mathit{multi}(r) && \{\text{fold } \mathit{delete}\ (57)\}\\
&= \mathit{multi}(\mathit{delete}(l, \mathit{some}(l))) \cup \{\mathit{some}(l)\} \cup \mathit{multi}(r) && \{\text{fold } \mathit{multi}\ (34)\}\\
&= \mathit{multi}(\langle \mathit{delete}(l, \mathit{some}(l)), \mathit{some}(l), r\rangle)
\end{aligned}$$

The choice function yields an arbitrary element of a non-empty search tree:

$$\begin{aligned}
& \mathit{some}\colon \mathcal{S}\backslash\{\varepsilon\} \to \mathcal{E}\\
& \mathit{some}(t) \in \mathit{multi}(t) = T
\end{aligned} \tag{61}$$

The choice is only restricted by the search tree property ($l \neq \varepsilon$):

$$\begin{aligned}
& \langle \mathit{delete}(l, \mathit{some}(l)), \mathit{some}(l), r\rangle \in \mathcal{S} && \{\text{by (44) and weaken conjunction}\}\\
\Rightarrow\ & \mathit{multi}(\mathit{delete}(l, \mathit{some}(l))) \sqsubseteq \mathit{some}(l) && \{\text{unfold } \mathit{delete}\ (57)\}\\
=\ & \mathit{multi}(l) \backslash \mathit{some}(l) \sqsubseteq \mathit{some}(l) && \{\text{by } (61)\}\\
=\ & \mathit{multi}(l) \sqsubseteq \mathit{some}(l)
\end{aligned}$$

Using the characterization (30) of the maximal element, this uniquely determines the choice: $\mathit{some} = \mathit{maxkey}$. It is easily derived that this equation is also sufficient for establishing the search tree property. Of course, symmetrically we could also delete an occurrence of the smallest element in the right subtree. In summary, the root deletion

$$\mathit{delroot}(\langle l, x, r\rangle) = \begin{cases} r & \text{if } l = \varepsilon\\ \langle \mathit{delmax}(l), \mathit{maxkey}(l), r\rangle & \text{if } l \neq \varepsilon\\ l & \text{if } r = \varepsilon\\ \langle l, \mathit{minkey}(r), \mathit{delmin}(r)\rangle & \text{if } r \neq \varepsilon \end{cases} \tag{62}$$

leads to two new subtasks. The function *delmax* deletes a maximal node from a non-empty search tree

$$\begin{aligned}
& \mathit{delmax}\colon \mathcal{S}\backslash\{\varepsilon\} \to \mathcal{S}\\
& \mathit{delmax}(t) = \mathit{delete}(t, \mathit{maxkey}(t))\,;
\end{aligned} \tag{63}$$

symmetrically $\mathit{delmin}\colon \mathcal{S}\backslash\{\varepsilon\} \to \mathcal{S}$ deletes a minimal one. Equation (62) again specifies a set of possible deletion functions. The remaining freedom can be exploited to meet further implementation constraints, for example to keep the tree balanced.

6.3 Deleting a Maximal/Minimal Element

We calculate a direct recursion for *delmax* (63), assuming $\langle l, x, r\rangle \in \mathcal{S}$.

$$\begin{aligned}
& \mathit{multi}(\mathit{delmax}(\langle l, x, r\rangle)) && \{\text{unfold } \mathit{delmax}\ (63)\}\\
= {} & \mathit{multi}(\mathit{delete}(\langle l, x, r\rangle, \mathit{maxkey}(\langle l, x, r\rangle))) && \{\text{unfold } \mathit{delete}\ (57),\ \mathit{maxkey}\ (51)\}\\
= {} & \mathit{multi}(\langle l, x, r\rangle) \backslash \mathit{maxel}(\mathit{multi}(\langle l, x, r\rangle)) && \{\text{unfold } \mathit{multi}\ (34)\}\\
= {} & \mathit{multi}(\langle l, x, r\rangle) \backslash \mathit{maxel}(\mathit{multi}(l) \cup \{x\} \cup \mathit{multi}(r)) && \{\text{by } (27)\}\\
= {} & \mathit{multi}(\langle l, x, r\rangle) \backslash \mathit{maxel}(\{x\} \cup \mathit{multi}(r)) = \ldots
\end{aligned}$$

Equation (26) now suggests the following case distinction:

Case $\mathit{multi}(r) = \emptyset$, that is $r = \varepsilon$.

$$\begin{aligned}
& = \mathit{multi}(\langle l, x, \varepsilon\rangle) \backslash x && \{\text{unfold } \mathit{multi}\ (34)\}\\
& = (\mathit{multi}(l) \cup \{x\} \cup \mathit{multi}(\varepsilon)) \backslash x && \{\text{by } (33),\ (2),\ (9)\}\\
& = \mathit{multi}(l) \cup (\{x\} \backslash x) && \{\text{by } (8)\}\\
& = \mathit{multi}(l) \cup \emptyset && \{\text{by } (2)\}\\
& = \mathit{multi}(l)
\end{aligned}$$

Case $\mathit{multi}(r) \neq \emptyset$, that is $r \neq \varepsilon$.

$$\begin{aligned}
& = \mathit{multi}(\langle l, x, r\rangle) \backslash \mathit{maxel}(\mathit{multi}(r)) && \{\text{fold } \mathit{maxkey}\ (48)\}\\
& = \mathit{multi}(\langle l, x, r\rangle) \backslash \mathit{maxkey}(r) && \{\text{unfold } \mathit{multi}\ (34)\}\\
& = (\mathit{multi}(l) \cup \{x\} \cup \mathit{multi}(r)) \backslash \mathit{maxkey}(r) && \{\text{by } (9)\}\\
& = \mathit{multi}(l) \cup \{x\} \cup (\mathit{multi}(r) \backslash \mathit{maxkey}(r)) && \{\text{fold } \mathit{delete}\ (57),\ \mathit{delmax}\ (63)\}\\
& = \mathit{multi}(l) \cup \{x\} \cup \mathit{multi}(\mathit{delmax}(r)) && \{\text{fold } \mathit{multi}\ (34)\}\\
& = \mathit{multi}(\langle l, x, \mathit{delmax}(r)\rangle)
\end{aligned}$$

In summary, a maximal node is deleted by successively visiting the right subtrees:

$$\mathit{delmax}(\langle l, x, r\rangle) = \begin{cases} l & \text{if } r = \varepsilon \\ \langle l, x, \mathit{delmax}(r)\rangle & \text{if } r \neq \varepsilon \end{cases} \tag{64}$$

Symmetrically, the deletion of a minimal node leads to

$$\mathit{delmin}(\langle l, x, r\rangle) = \begin{cases} r & \text{if } l = \varepsilon \\ \langle \mathit{delmin}(l), x, r\rangle & \text{if } l \neq \varepsilon\,. \end{cases} \tag{65}$$

It is straightforward to show that *delmax* (64) and *delmin* (65) maintain the binary search tree property.

6.4 Combining the Subdevelopments

When combining the subdevelopments from Sections 6.1 to 6.3, we expand the auxiliary routine *delroot* (62) and obtain from (58)–(59)

$$\mathit{delete}(\varepsilon, y) = \varepsilon \tag{66}$$

$$\mathit{delete}(\langle l, x, r\rangle, y) = \begin{cases} \langle \mathit{delete}(l, y), x, r\rangle & \text{if } y < x \\ r & \text{if } y = x \wedge l = \varepsilon \\ l & \text{if } y = x \wedge r = \varepsilon \\ \langle \mathit{delmax}(l), \mathit{maxkey}(l), r\rangle & \text{if } y = x \wedge l \neq \varepsilon \\ \langle l, \mathit{minkey}(r), \mathit{delmin}(r)\rangle & \text{if } y = x \wedge r \neq \varepsilon \\ \langle l, x, \mathit{delete}(r, y)\rangle & \text{if } y > x\,. \end{cases} \tag{67}$$

Again, the remaining choice in equation (67) can be exploited to meet further implementation constraints. A uniform choice taking at every node for example the left subtree, yields

$$delete(\varepsilon, y) = \varepsilon \tag{68}$$

$$delete(\langle l, x, r\rangle, y) = \begin{cases} \langle delete(l, y), x, r\rangle & \text{if } y < x \\ r & \text{if } y = x \wedge l = \varepsilon \\ \langle delmax(l), maxkey(l), r\rangle & \text{if } y = x \wedge l \neq \varepsilon \\ \langle l, x, delete(r, y)\rangle & \text{if } y > x\,. \end{cases} \tag{69}$$

6.5 Merging Multiple Tree Traversals

The function *delete* (68)–(69) traverses the successive left subtrees twice: once for calculating the greatest element and once more for deleting it. Hence we merge the two subtasks *delmax* and *maxkey* into a single function *delmaxkey* using function tupling:

$$\begin{aligned} &delmaxkey\colon \mathcal{S}\backslash\{\varepsilon\} \to \mathcal{S} \times \mathcal{E} \\ &delmaxkey = [delmax, maxkey] \end{aligned} \tag{70}$$

It is now straightforward to derive a direct recursion:

Induction basis

$$\begin{aligned} & delmaxkey(\langle l, x, \varepsilon\rangle) && \{\text{unfold } delmaxkey\ (70)\} \\ =\ & (delmax(\langle l, x, \varepsilon\rangle), maxkey(\langle l, x, \varepsilon\rangle)) && \{\text{unfold } delmax\ (64),\ maxkey\ (53)\} \\ =\ & (l, x) \end{aligned}$$

Induction step, assuming $r \neq \varepsilon$.

$$\begin{aligned} & delmaxkey(\langle l, x, r\rangle) && \{\text{unfold } delmaxkey\ (70)\} \\ =\ & (delmax(\langle l, x, r\rangle), maxkey(\langle l, x, r\rangle)) && \{\text{unfold } delmax\ (64),\ maxkey\ (53)\} \\ =\ & (\langle l, x, delmax(r)\rangle, maxkey(r)) && \{\text{introduce constants } s \in \mathcal{S}, k \in \mathcal{E}\} \\ =\ & \lceil (s, k) = (delmax(r), maxkey(r));\ (\langle l, x, s\rangle, k)\rfloor && \{\text{fold } delmaxkey\ (70)\} \\ =\ & \lceil (s, k) = delmaxkey(r);\ (\langle l, x, s\rangle, k)\rfloor \end{aligned}$$

This yields the final version

$$delete(\varepsilon, y) = \varepsilon \tag{71}$$

$$delete(\langle l, x, r\rangle, y) = \begin{cases} \langle delete(l, y), x, r\rangle & \text{if } y < x \\ r & \text{if } y = x \wedge l = \varepsilon \\ \langle delmaxkey(l), r\rangle & \text{if } y = x \wedge l \neq \varepsilon \\ \langle l, x, delete(r, y)\rangle & \text{if } y > x \end{cases} \tag{72}$$

where

$$delmaxkey(\langle l, x, r\rangle) = \begin{cases} (l, x) & \text{if } r = \varepsilon \\ \lceil (s, k) = delmaxkey(r);\ (\langle l, x, s\rangle, k)\rfloor & \text{if } r \neq \varepsilon\,. \end{cases} \tag{73}$$

Both functions *delete* and *delmaxkey* show a linear, but non-tail recursion.

6.6 Realization

This completes the calculation of the functional module

$$(\mathcal{S};\, \varepsilon,\, \mathit{find},\, \mathit{minkey},\, \mathit{maxkey},\, \mathit{insert},\, \mathit{delete})$$

for binary search trees. The binary search tree property is guaranteed only for those trees that are generated from the empty tree ε using the functions *insert* and *delete* . Hence the representation of the search trees has to be encapsulated and the ternary tree constructor $\langle .,.,. \rangle$ must not be provided to the user of the functional module.

The algorithms derived can directly be transcribed into a functional programming language. Heading towards an implementation on the VON NEUMANN machine, in the next step a pointer structure for binary search trees can safely be introduced using pointer algebra [Mö97].

7 Reusing the Development

In the formal development of the search tree module we have jointly derived from the specification an efficient algorithmic solution together with an implementation of the data structure. In this section we show how *redesign* techniques are well supported by transformational programming.

There is an increasing demand for reusing software and adapting it to changing requirements. However, it is difficult to modify existing code to meet a changing specification. If the design of an algorithm is documented by a program derivation, then the development can often be "replayed" starting with a slightly modified specification. As an example, we "recalculate" the algorithm for a different basic data type with a similar set of laws.

7.1 Redesign

We consider the related problem of implementing binary search trees with pairwise different elements which can be used as keys. Then the search tree represents a *set* (and not a multiset) of elements.

For the algebra $(\mathcal{S}; \emptyset, \{.\}, .\cup., .\backslash., . \in .)$ of finite sets, equations (1)–(8) from Section 2.1 hold. The set union satisfies the additional law

$$m \cup m = m\,, \tag{74}$$

and equation (9) simplifies to

$$(m \cup n)\backslash y = (m\backslash y) \cup (n\backslash y)\,. \tag{75}$$

Moreover, one can show relations analogous to (19)–(30) using the abstraction function $\mathit{set}\colon \mathcal{T} \to \mathcal{S}$ defined by

$$\mathit{set}(\varepsilon) = \emptyset \tag{76}$$

$$\mathit{set}(\langle l, x, r\rangle) = \mathit{set}(l) \cup \{x\} \cup \mathit{set}(r)\,, \tag{77}$$

compare (33)–(34). The derivation of the search function then leads to *repetition-free search trees* $\mathcal{R} \subseteq \mathcal{T}$ inductively generated by

$$(i)\ \ \varepsilon \in \mathcal{R} \tag{78}$$

$$(ii)\ \ \text{If } l, r \in \mathcal{R} \text{ and } x \in \mathcal{E} \text{ with } set(l) \sqsubset x \sqsubset set(r),\ \text{then } \langle l, x, r\rangle \in \mathcal{R}\,, \tag{79}$$

compare (43)–(44). The remaining derivations of the functions *find*, *minkey* and *maxkey* do not change, since they are only based on the containment and order relation. The replayed derivation of the function *insert* yields

$$insert(\varepsilon, y) = \langle \varepsilon, y, \varepsilon\rangle \tag{80}$$

$$insert(\langle l, x, r\rangle, y) = \begin{cases} \langle insert(l, y), x, r\rangle & \text{if } y < x \\ \langle l, x, r\rangle & \text{if } y = x \\ \langle l, x, insert(r, y)\rangle & \text{if } y > x\,, \end{cases} \tag{81}$$

compare (55)–(56). Here the additional case distinction arises from using equation (74) in the induction step. Similarly, the derivation of the function *delete* can be replayed for repetition-free search trees; it leads with slightly different reasoning to the same result (71)–(73) as for search trees.

In this way the expense of a formal program development pays, since it provides a wide spectrum of related algorithms as a by-product. This allows replacing the textual modification of existing programs — which is current practice — by a semantics-based redevelopment.

7.2 Refinement

If a program is derived under weak assumptions, then the development refines the specification in each step only gradually and no more than necessary. The remaining algorithmic freedom can be exploited to impose further implementation constraints in order to improve efficiency. As an example we revisit the search function (38)–(39)

$$search(\varepsilon, y) = F$$

$$search(\langle l, x, r\rangle, y) = \begin{cases} T & \text{if } y = x \\ search(l, y) \vee search(r, y) & \text{if } y \neq x \end{cases}$$

from Section 3.1. The simplification of the cascading recursion to a linear recursion gave rise to search trees. A different efficiency improvement consists in limiting the maximal recursion depth of the search function. However, the recursion depth of the function *search* (38)–(39) corresponds to the height of the tree:

$$height\colon\ \mathcal{T} \to \mathcal{N}$$

$$height(\varepsilon) = 0 \tag{82}$$

$$height(\langle l, x, r\rangle) = 1 + max(height(l), height(r))\,. \tag{83}$$

The design decision to limit the recursion depth by constraining the data structure leads to the set $\mathcal{H}$ of *(height-)balanced trees* inductively generated by

$$(i)\ \ \varepsilon \in \mathcal{H} \tag{84}$$

$$(ii)\ \ \text{If } l, r \in \mathcal{H} \text{ and } x \in \mathcal{E} \text{ with } |height(l) - height(r)| \leq 1, \text{ then } \langle l, x, r\rangle \in \mathcal{H}\,, \tag{85}$$

compare [AL62]. On the subset of height-balanced search trees the corresponding algorithms (see, for example, [AHU83, CLR90, K73, Mh84, S88]) can be derived in a similar, yet more involved manner as for simple search trees.

8 Conclusion

There is a large conceptual gap between an abstract problem-oriented specification and a fully detailed, efficiently executable machine-oriented program. Transformational programming replaces the traditional "programming in one blow" by a series of small, formally controllable, semantics-preserving steps, each adding details through design decisions. In this way transformational programming guarantees the correctness of the final program with respect to the initial specification.

The development gives insight into the algorithmic principles underlying a program: a particular implementation is not understood through its code but by the design decisions that lead to it. When developing a family tree of different algorithms from the same specification, the derivation relates the different implementations which cannot be compared at the code level. The transformation rules as well as the development strategies formalize programming knowledge. The calculational nature of program manipulation also supports building program development systems (for an overview see [F87]).

Functional programming lends itself particularly well to this purpose: its expressions are manipulated with the same ease as conventional, mathematical expressions. Moreover, frequently the data and control structures involved obey a rich set of algebraic laws. This eliminates much of the burden of manipulating quantifiers explicitly and leads to concise and perspicuous derivations.

So there is hope that in the long run formal program development techniques will play the same rôle for software engineering as mathematics plays in traditional engineering disciplines.

Acknowledgements We thank F. Nickl and M. Russling for valuable discussions, C. Runciman for a pointer to the literature, and the anonymous referees for helpful comments.

References

[AL62] G.M. Adelson-Velskii, Y.M. Landis: *An Algorithm For the Organisation of Information.* Doklady Akademia Nauk SSR *146*, 263–266 (1962). English translation: Soviet Math. *3*, 1259–1263

[AHU83] A.V. Aho, J.E. Hopcroft, J.D. Ullman: *Data Structures and Algorithms.* Reading, Mass.: Addison-Wesley 1983

[BMPP89] F.L. Bauer, B. Möller, H. Partsch, P. Pepper: *Formal Program Construction by Transformations — Computer-aided, Intuition-guided Programming.* IEEE Transactions on Software Engineering *15*, 165–180 (1989)

[B89] R. Bird: *Lectures on Constructive Functional Programming.* In: M. Broy (ed.): Constructive Methods in Computer Science. NATO ASI Series F: Computer and Systems Sciences *55.* Berlin: Springer 1989, 151–216

[CLR90] T.H. Cormen, C.E. Leiserson, R.L. Rivest: *Introduction to Algorithms.* The MIT Electrical Engineering and Computer Science Series. Cambridge, Mass.: M.I.T. Press/New York: McGraw-Hill 1990

[F87] M. Feather: *A Survey and Classification of Some Program Transformation Approaches and Techniques.* In: L.G.L.T. Meertens (ed.): Proceeding of the IFIP TC2/WG2.1 Working Conference on Program Sepcification and Transformation. Amsterdam: North-Holland 1987, 165–196

[HD83] C.M. Hoffman, M.J. O'Donnell: *Programming with Equations.* ACM Transaction on Programming Languages and Systems *4*:6, 83–112 (1983)

[H89] P. Hudak: *Conception, Evolution, and Application of Functional Programming Languages.* ACM Computing Surveys *21*:3, 359–411 (1989)

[K73] D.E. Knuth: *The Art of Computer Programming.* Vol. *3*: Sorting and Searching. Reading, Mass.: Addison-Wesley 1973

[Me86] L.G.L.T. Meertens: *Algorithmics — Towards Programming as a Mathematical Activity.* Proceedings CWI Symposium on Mathematics and Computer Science. CWI Monographs Vol. *1.* Amsterdam: North–Holland 1986, 289–334

[Mh84] K. Mehlhorn: *Data Structures and Algorithms 1: Sorting and Searching.* EATCS Monographs in Theoretical Computer Science. Berlin: Springer 1984

[Mö97] B. Möller: *Calculating with pointer structures*. In: R. Bird, L. Meertens (eds.): Algorithmic Languages and Calculi. Proc. IFIP TC2/WG2.1 Working Conference, Le Bischenberg, Feb. 1997. Chapman&Hall 1997 (to appear)

[P90] H.A. Partsch: *Specification and Transformation of Programs — A Formal Approach To Software Development.* Berlin: Springer 1990

[R92] C.M.P. Reade: *Balanced Trees with Removals: An Exercise in Rewriting and Proof.* Science of Computer Programming *18*, 181–204 (1992)

[S88] R. Sedgewick: *Algorithms.* Reading, Mass.: Addison-Wesley 1988