AVL Trees Revisited: A Case Study in SPECTRUM

R. Hettler, D. Nazareth, F. Regensburger, O. Slotosch

Institut für Informatik
Technische Universität München
Arcisstr. 21, D-80290 München
e-mail: {hettler,nazareth,regensbu,slotosch}@informatik.tu-muenchen.de

August 1994

Abstract

The goal of this paper is to present a way of developing software within the axiomatic specification language SPECTRUM. We advocate a development approach which is organized in four stages: requirement specification, design specification, executable specification and functional program. The concept of AVL trees serves as example for presenting this approach.

Chapter 1

Introduction

This paper presents an example for a complete software development in the specification language SPECTRUM. SPECTRUM is an axiomatic specification language supporting full first-order logic including some higher-order concepts. Its loose semantics allows for stepwise development because it is possible to refine specifications by adding properties in terms of additional axioms. This has the effect that the class of models of the refined specification is always included in the class of models of the original one. Features like a polymorphic type system, higher-order functions and built-in λ-abstraction make it especially suited for developing functional programs. For a detailed description of SPECTRUM see [BFG+93a, BFG+93b].

As topic for this case study we chose AVL trees [AVL62, Wir76] for the following reasons:

- It is small enough to present the whole development together with the motivation for the single development steps and with all verification conditions.
- It is a well-known subject which already has been dealt with in many case studies in different formalisms (see for example [Sch90]). Therefore we can concentrate on the SPECTRUM specifications and point out how they are used to support the different development phases.

By means of this example we show the typical stages a SPECTRUM development goes through: requirement specification, design specification, executable specification and program. In our example the implementation language is Gofer [Jon93], a dialect of the lazy functional language Haskell [HJW92]. This language was chosen because it has a type system with type classes which is very similar to SPECTRUM's sort system. This ensures that the transition from SPECTRUM to the implementation language is smooth and understandable.

This paper is structured as follows. All the basic concepts we use in specifying AVL trees are presented in Section 2. Section 3 discusses the main features of requirement specifications and gives a concrete requirement specification for AVL trees. In Section 4 we develop the requirement specification into a design specification by bringing in the

algorithmic idea of rotations of subtrees. These rotations are used in the insert and delete operations on AVL trees to keep them balanced. We also present the verification conditions which are necessary to show the implementation relation between the different specification stages. Because of the limited space allowed for this paper we do not present the detailed proofs. The notion of an executable specification is then dealt with in Section 5. We show how the design specification can be developed into a specification that is executable in the sense of our implementation language Haskell. This is where the specific details of the implementation language are brought in. From this executable specification the Haskell program can be derived by a simple syntactic transformation as is shown in Section 6. Finally, in Section 7 we discuss the whole development and especially the role of SPECTRUM.

Chapter 2

Prerequisites

Our specification of AVL trees is based on some fundamental concepts which we will list in this section. For height calculations on our trees, we need the concept of natural numbers with a total order $.\leq.$ defined on them. For this we can refer to SPECTRUM's standard library (see [BFG+93b]). The following specification enriches the specification `Naturals` from the standard library by a function `max` calculating the maximum of two natural numbers:

```
Nat = {enriches Naturals;

   max : Nat × Nat → Nat;
   max strict total;

   axioms ∀ n,m:Nat in
     max(n,m) = if n ≤ m then m else n endif;
   endaxioms;
}
```

Furthermore we will make use of the concept of binary trees because AVL trees are a special case of ordered binary trees[1]:

```
Bintree = {enriches Nat;

   data Tree α = emptytree
               | mktree(!node:α, !left:Tree α, !right:Tree α);
   Tree :: (EQ)EQ;

   height : Tree α → Nat;
   .isin. : α::EQ ⇒ α × Tree α → Bool                    prio 6;
   height, .isin. strict total;
```

[1]The exclamation marks used in the `data` declaration require the constructor function `mktree` to be strict in all arguments.

```
   axioms ∀ n:α, l,r:Tree α in
      height(emptytree) = 0;
      height(mktree(n,l,r)) = succ(max(height(l),height(r)));
   endaxioms;

   axioms α::EQ ⇒ ∀ a,n:α, l,r:Tree α in
      ¬(a isin emptytree);
      a isin (mktree(n,l,r)) = (a == n ∨ a isin l ∨ a isin r);
   endaxioms;
}
```

The concept of ordered binary trees is specified as a predicate on binary trees. Of course, we can only define ordered trees on elements on which a total order is given. Therefore the following specification refers to the specification `Ordering` from the standard library, in which the sort class `TO` of sorts with a total order is specified.

```
Ordtree = {enriches Bintree + Ordering;

   .<. : α::TO ⇒ α × α → Bool                    prio 6;
   isord : α::TO ⇒ Tree α  → Bool;
   .<., isord strict total;

   axioms α::TO ⇒ ∀ n,m:α, l,r:Tree α in
     m < n = ((m ≤ n) ∧ ¬(m == n));

     isord(emptytree);
     isord(mktree(n,l,r)) = ((∀n'. n' isin l ⇒ n' < n) ∧
                             (∀n'. n' isin r ⇒ n < n') ∧
                             isord(l) ∧ isord(r));
   endaxioms;
}
```

Chapter 3

Requirement Specification

Given the prerequisites of Section 2 we can now concentrate on the requirement specification of AVL trees. A requirement specification describes at an abstract level a problem domain and the essential functional requirements that every solution of the problem has to fulfill. It can (even should) be purely descriptive and need not give an algorithm for the solution. The problem needs not be described completely, decisions which are not essential can be postponed to a later development phase (underspecification). The only way in which a requirement specification should take into account the topic of efficiency is that it should not prevent possible efficient solutions (e.g. by unlucky axiomatization).

An AVL tree is a binary ordered tree that fulfills the following balance criterion:

> A binary tree `t` is *balanced* iff for every node `n` of `t` the heights of the two subtrees of `n` differ by at most one.

As with ordered trees AVL trees are axiomatized in SPECTRUM by giving predicates for checking both the balance criterion and the complete AVL criterion on binary trees. Furthermore, functions for inserting and deleting elements in AVL trees are specified which preserve the AVL property.

```
Avltree_req = {enriches Ordtree;

isbal : Tree α → Bool;
isbal strict total;

axioms ∀ n:α, l,r:Tree α in
   isbal(emptytree);
   isbal(mktree(n,l,r)) =
     (isbal(l) ∧ isbal(r) ∧
     (height(l) == height(r) ∨ height(l) == succ(height(r))
      ∨ succ(height(l)) == height(r)));
endaxioms;

isavl  : α::TO ⇒ Tree α → Bool;
```

```
insert : α::TO ⇒ α × Tree α → Tree α;
delete : α::TO ⇒ α × Tree α → Tree α;
isavl, insert, delete strict total;

axioms α::TO ⇒  ∀ x,y:α, t:Tree α in
      isavl(t) = (isord(t) ∧ isbal(t));
{ins} isavl(t) ⇒ isavl(insert(x,t)) ∧
                  (y isin insert(x,t) = (x=y ∨ y isin t));
{del} isavl(t) ⇒ isavl(delete(x,t)) ∧
                  (¬(y isin delete(x,t)) = (x=y ∨ ¬(y isin t)));
endaxioms;
}
```

Chapter 4

Development of a Design Specification

The goal of a design specification is to introduce algorithms which solve the functional requirements described in the requirement specification. In contrast to the executable specification presented in Section 5 the design specification does not cope with technical details of the selected implementation language. Further it does not need to fulfil the syntactic restrictions of an executable specification (see Section 5.1).

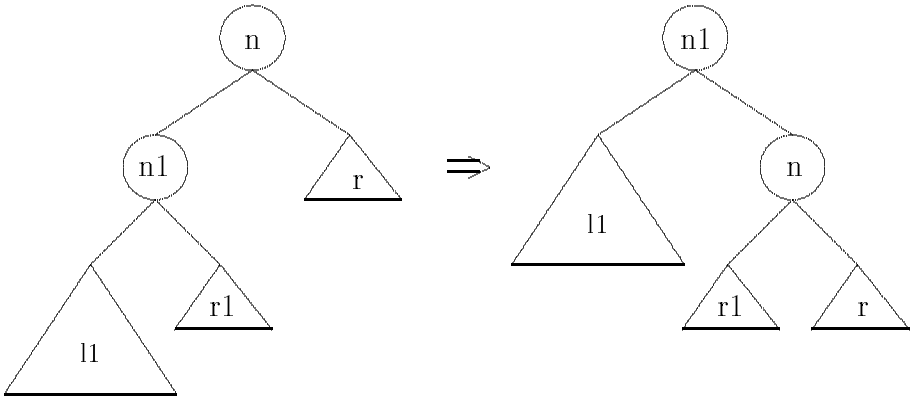
Normally the design specification is developed from the requirement specification in several steps. The specification is made more and more detailed and algorithmic. For this purpose the specification is restructured and enriched by additional axioms. Axioms which are less constructive are proved to be theorems of the enriched specification and deleted. Furthermore, auxiliary sorts and functions can be introduced. Our example is small enough to do this in one development step.

Of course, the developed design specification has to comply with the requirement specification from Section 3. We have to show that the design specification indeed is an implementation of the requirement specification. The particular proof obligations are discussed in Section 4.4 after presenting the design specification.

In Section 3 we specified the balance criterion for our search trees. If we insert and delete in a balanced search tree this balance criterion can be violated. If it is violated the search tree must be restructured to restore balance. For reasons of efficiency restructuring is done during the process of insertion and deletion. Before specifying the algorithm we explain both algorithms informally.

4.1 Insertion

Every insertion of a node in balanced trees could lead to a violation of the AVL criterion. Given a node with the left and right subtrees L and R, three cases must be distinguished. Assume that a new node is inserted in L causing its height to increase by 1:



rightrotation(n,mktree(n1,l1,r1),r) = mktree(n1,l1,mktree(n,r1,r))

Figure 4.1: rightrotation

1. height(L) = height(R): The heights of L and R become unequal, but the balance criterion is not violated.

2. height(L) < height(R): The heights of L and R become equal, the balance has even been improved.

3. height(L) > height(R): After insertion the height of L and R differ by 2. Therefore the balance criterion becomes violated, and the tree must be restructured.

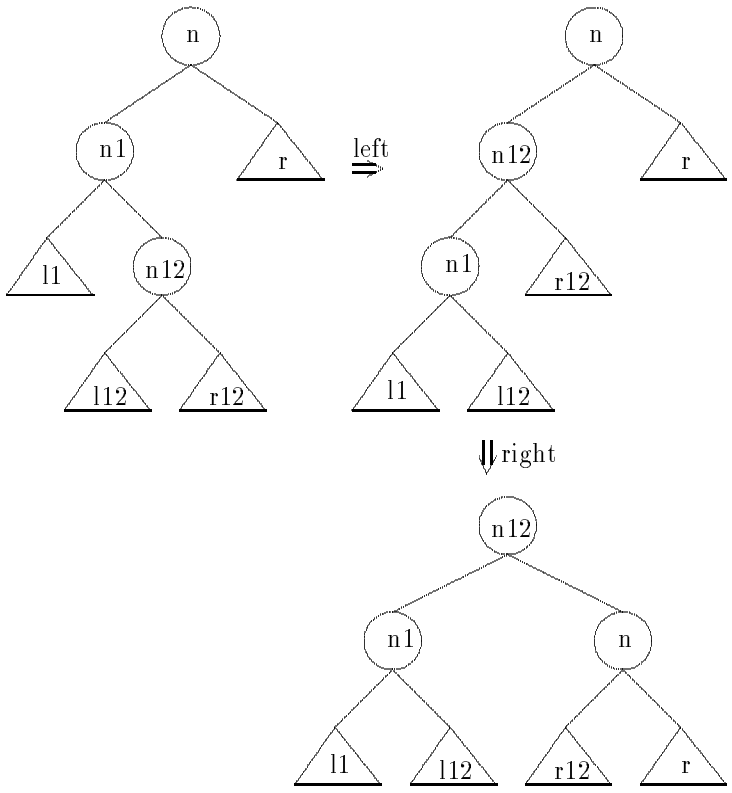
Only in the third case the tree must be rebalanced. To restore balance, nodes are cyclically exchanged, depending on the balance of the left subtree, resulting in either a single or a double rotation of the two or three nodes involved. Both rotations preserve the ordering of the tree elements. The specification of the algorithm is shown in the design specification `Avltree_des` in Section 4.3. The single rotation is done by the function `rightrotation` and the double rotation by the function `leftrightrotation`. In Figure 4.1 and 4.2 the principle of both functions is illustrated. The terms below the figures are according to the axioms `rrot` and `lrrot` of the design specification.

If we insert a new node in the right subtree R the balance criterion can also be violated, because R possibly becomes too high. Then we need the functions `leftrotation` and `rightleftrotation` to rebalance the tree, which work in a similar way. For a more detailed description of the rotation algorithms see [Wir76].

The process of node insertion consists essentially of the following three consecutive parts:

1. Follow the search path until it is verified that the key is not already in the tree.

2. Insert the new node



leftrightrotation(n,mktree(n1,l1,mktree(n12,l12,r12)),r) =
mktree(n12,mktree(n1,l1,l12),mktree(n,r12,r))

Figure 4.2: leftrightrotation

3. Go back along the search path and check the balance at each node.

This method involves some redundant checking. Once balance is established, it needs not to be checked on that node's predecessors. The specification of the insert algorithm is presented in the design specification `Avltree_des` in Section 4.3.

The working principle is shown on an example from [Wir76] in Figure 4.3. Consider the binary tree (a) which consists of two nodes. Insertion of key 7 first results in an unbalanced tree (i.e. a linear list). Its balancing involves a single leftrotation, resulting in the perfectly balanced tree (b). Further insertion of keys 2 and 1 result in an imbalance of the subtree with root node 4. This subtree is balanced by a single rightrotation (d). The subsequent insertion of key 3 immediately violates the balance criterion at the root node 5. Balance is thereafter re-established by a more complicated leftrightrotation (e). The only candidate for loosing balance after a next insertion is node 5. Indeed, insertion of node 6 must invoke the fourth case of rebalancing outlined, the rightleftrotation. The final tree is shown in Figure 4.3 (f).

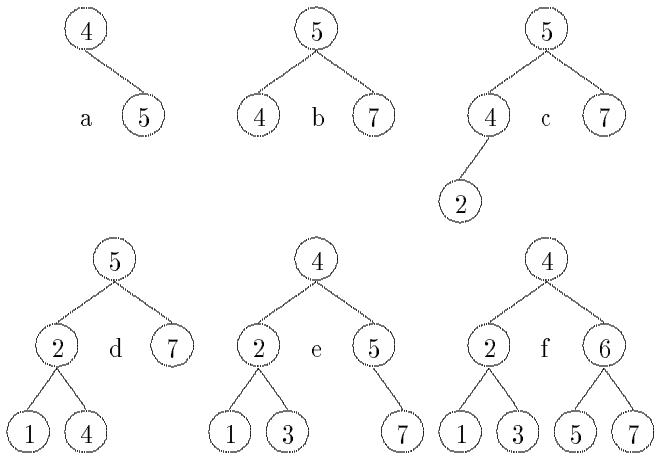


Figure 4.3: Insertions in a balanced tree

4.2 Deletion

Deletion in balanced trees is more complicated than insertion. The easy cases to delete are leaves or nodes with only one single descendant. If the node to be deleted has two subtrees, we will replace it by the rightmost leaf of its left subtree and delete this node. Like insertion of a node, deletion can violate the balance criterion too. Fortunately the process of rebalancing remains the same. However, in contrast to insertion the balance criterion can not only be violated once during deletion, but successively at each node. The algorithm is again specified in the design specification `Avltree_des` in Section 4.3.

The working principle is shown on an example from [Wir76] in Figure 4.4. Given the balanced tree (a), successive deletion of the nodes with the keys 4, 8, 6, 5 and 2 results in the trees (b)–(f). The deletion of key 4 is simple since it represents a terminal node. However, it results in an unbalanced node 3. Its rebalancing operation involves a single rightrotation. Rebalancing becomes again necessary after the deletion of node 6. This time the right subtree of the root 7 is rebalanced by a single leftrotation. Deletion of node 2, although in itself straightforward since it has only a single descendant, makes a complicated rightleftrotation necessary.

4.3 Design Specification

Now we give a design specification `Avltree_des` that formalizes the algorithms described in the last two sections. The specification is based on a specification `Ordtree_des` providing all algorithms for the primitive functions. We omit this specification and the resulting proof obligations. Note that in contrast to the algorithms presented in [Wir76] our algorithms for `insert` and `delete` do not have complexity O(log n), where n is the number of elements in the tree. For reasons of simplicity we do not store the height of each subtree in the nodes

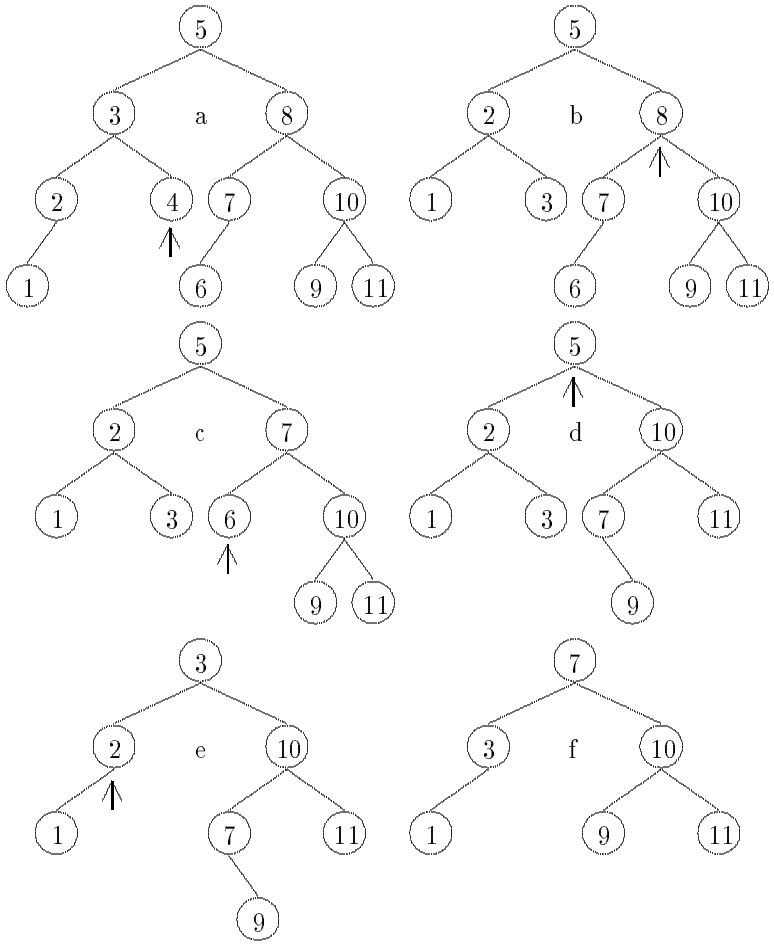


Figure 4.4: Deletions in a balanced tree

of the tree. The height is always computed on demand. The more efficient version would require a change of the data structure to store this information.

```
Avltree_des = { enriches Ordtree_des;
    strict;

    -- auxiliary sort
    data Balance = just | leftweight | rightweight;
    Balance :: EQ;

    isbal : Tree α → Bool;

    axioms ∀ n:α, l,r:Tree α in

        isbal(emptytree);
```

```
    isbal(mktree(n,l,r)) = (isbal(l) ∧ isbal(r) ∧
        (height(l) == height(r) ∨
         height(l) == succ(height(r)) ∨
         succ(height(l)) == height(r)));

endaxioms;

isavl : α::TO ⇒ Tree α → Bool;
insert : α::TO ⇒ α × Tree α → Tree α;
delete : α::TO ⇒ α × Tree α → Tree α;

-- auxiliary functions
balance : Tree α → Balance;
rightrotation : α × Tree α × Tree α → Tree α;
leftrotation : α × Tree α × Tree α → Tree α;
rightleftrotation : α × Tree α × Tree α → Tree α;
leftrightrotation : α × Tree α × Tree α → Tree α;
leftbalance : α × Tree α × Tree α → Tree α;
rightbalance : α × Tree α × Tree α → Tree α;
grel : Tree α → α;

-- specification of auxiliary functions
axioms ∀ n,n1,n2,n21,n12 : α,
         l,r,l1,r1,l2,r2,l21,r21,l12,r12 : Tree α in

    -- balance computes the balance of a node
    balance(emptytree) = just;
    balance (mktree(n,l,r)) =
        if height l == height r then just
        else if succ(height l) ≤ height r then rightweight
             else leftweight
             endif
        endif;

{rrot}  rightrotation(n,mktree(n1,l1,r1),r) =
        mktree(n1,l1,mktree(n,r1,r));
{lrot}  leftrotation(n,l,mktree(n2,l2,r2)) =
        mktree(n2,mktree(n,l,l2),r2);
{rlrot} rightleftrotation(n,l,mktree(n2,mktree(n21,l21,r21),r2)) =
        mktree(n21,mktree(n,l,l21),mktree(n2,r21,r2));
{lrrot} leftrightrotation(n,mktree(n1,l1,mktree(n12,l12,r12)),r) =
        mktree(n12,mktree(n1,l1,l12),mktree(n,r12,r));

    -- the functions leftbalance and rightbalance restore balance
    -- at a node by performing a simple or a double rotation
```

```
    leftbalance(n,l,r) =
        if balance(l) == rightweight then leftrightrotation(n,l,r)
            else rightrotation(n,l,r)
        endif;
    rightbalance(n,l,r) =
        if balance(r) == leftweight then rightleftrotation(n,l,r)
            else leftrotation(n,l,r)
        endif;

    -- grel computes the greatest element of an ordered binary tree
    grel(mktree(n,l,emptytree)) = n;
    grel(mktree(n,l,mktree(n1,l1,r1))) = grel(mktree(n1,l1,r1));
endaxioms;

-- specification of insert and delete
axioms α :: T0 ⇒ ∀ x,n,n1,n2 :α, t,l,r,l1,r1,l2,r2 : Tree α in

    isavl(t) = (isord(t) ∧ isbal(t));

    insert(x,emptytree) = mktree(x,emptytree,emptytree);
    insert(x,mktree(n,l,r)) =
        if x == n then mktree(n,l,r)
        else if x < n then
                let l2 = insert(x,l)in
                    if height(l2) == succ(succ(height(r))) then
                        leftbalance(n,l2,r)
                    else mktree(n,l2,r)
                    endif
                endlet
             else let r2 = insert(x,r) in
                    if height(r2) == succ(succ(height(l))) then
                        rightbalance(n,l,r2)
                    else mktree(n,l,r2)
                    endif
                  endlet
             endif
        endif;

{de_del1} delete(x,emptytree) = emptytree;
{de_del2} delete(x,mktree(n,l,emptytree)) =
        if x == n then l
        else if x < n then
                 mktree(n,delete(x,l),emptytree)
             else mktree(n,l,emptytree)
             endif
```

```
        endif;
    {de_del3} delete(x,mktree(n,emptytree,r)) =
        if x == n then r
        else if n < x then
                 mktree(n,emptytree,delete(x,r))
             else mktree(n,emptytree,r)
             endif
        endif;
    {de_del4} delete(x,mktree(n,mktree(n1,l1,r1),mktree(n2,l2,r2))) =
        if x == n then
            let gr = grel(mktree(n1,l1,r1)) in
                let ll = delete(gr,mktree(n1,l1,r1)) in
                    if height(mktree(n2,l2,r2)) ==
                            succ(succ(height(ll))) then
                        rightbalance(gr,ll,mktree(n2,l2,r2))
                    else mktree(gr,ll,mktree(n2,l2,r2))
                    endif
                endlet
            endlet
        else if x < n then
                let ll = delete(x,mktree(n1,l1,r1)) in
                    if height(mktree(n2,l2,r2)) ==
                            succ(succ(height(ll))) then
                        rightbalance(n,ll,mktree(n2,l2,r2))
                    else mktree(n,ll,mktree(n2,l2,r2))
                    endif
                endlet
             else
                 let rr = delete(x,mktree(n2,l2,r2))in
                     if height(mktree(n1,l1,r1)) ==
                             succ(succ(height(rr))) then
                         leftbalance(n,mktree(n1,l1,r1),rr)
                     else mktree(n,mktree(n1,l1,r1),rr)
                     endif
                 endlet
             endif
        endif;
    endaxioms;
}
```

4.4 Proof Obligations

Up to now we presented a requirement specification and a completely independent design specification. It remains to show that the design specification provides indeed one of

many possible implementations for the `insert` and `delete` function of the requirement specification. We must show that the algorithms fulfil the axioms we demanded in the requirement specification.

Both the insert and delete function are not straightforward. It costs a bit of time to understand how the rotations are rebalancing the search tree during insertion and deletion and therefore restore the balance criterion. Further, such relatively complicated algorithms are always susceptical to typing errors. Therefore only the use of a formal proof system can guarantee the correctness of the algorithms with respect to the requirement specification.

The balance and AVL criterion was already described constructively in the requirement specification and did not change. However, the very abstract axioms `ins` and `del` from the requirement specification `Avltree_req` in Section 3 have been omitted in the design specification. Therefore we must prove that both `ins` and `del` are theorems of the design specification.

Besides this obvious proof obligation there are more hidden ones to be proved. In the requirement specification we defined the functions `isbal`, `isavl`, `insert` and `delete` to be totally defined by writing e.g. `isbal total`. This is a requirement to the algorithm of the functions to yield a defined value on defined arguments. These totality axioms have been omitted in the design specification because no programming language can guarantee this behaviour. Therefore, in contrast to strictness, ensuring totality is independent of the selected implementation language and has to be done in the design specification. Since `isbal total` is a shortcut for the axiom $\forall^{\perp}$`x:Tree` α. δ`(x)` $\Rightarrow$ δ`(isbal(x))` we have to prove this axiom to be a theorem of the design specification. The same has to be done for the functions `isavl`, `insert` and `delete`.

Due to reasons of space, we can not present the proofs in this paper. The proofs were carried out using the proof system Isabelle [Pau93] in the form of a diploma thesis [Pus94].

4.5 Sort Classes

In Haskell, if a sort is defined to be an element of a particular class, all member functions must be instantiated by concrete implementations for this sort. In a specification language like SPECTRUM this proceeding would in general be too restrictive, because the instance is only needed to execute the program. Specifying a sort to belong to a particular class does not require an instance for any function[1]. It only demands that all axioms valid for the elements of the class sorts must be valid for the elements of the given sort. This is a very abstract view of classes and by no means algorithmic. However, if we want to develop an executable specification we must for every sort belonging to a class provide an executable instantiation for each member function. Because the whole treatment of classes is fairly dependent of the selected implementation language, we postpone the specification of implementations and the explicit instantiation to Section 5.

[1] In fact, SPECTRUM does neither provide a syntactic construct to declare member functions, nor to instantiate member functions.

Chapter 5

Development of an Executable Specification

Software development is a long process. The requirement specification is an abstract specification of the problem. The design specification contains algorithms which solve the problem. But software development is more: It should result in an executable program. There are many different programming languages and we have to choose one as target language for our development. For this target language there is a corresponding part of the SPECTRUM specification language marked out such that all specifications written in this *corresponding executable part* of SPECTRUM can be directly translated into programs of the target language. We call such specifications *executable specifications*. The choice of the target programming language is important, since it influences the resulting program and its development process. If we choose a simple (and efficient) language we have a smaller corresponding executable part and therefore we have a longer development process until the specification is executable. If we choose a more complex language the development will be shorter, since the corresponding executable part is larger. The software development continues until the specification is completely written in the executable part. Of course during the development the correctness of all steps has to be proved.

In our case we decided to work with the programming language Haskell and its efficient implementation Gofer [Jon93]. First we give a short description of SPECTRUM's executable part corresponding to Gofer. Then we develop an executable specification and give the proof obligations which imply the correctness of this development step.

5.1 Gofer's corresponding Specifications

In the following we will use the term 'executable specification' instead of 'SPECTRUM's executable part corresponding to Gofer'. This means that executable specifications can be translated directly into Gofer programs. Now we will give some important syntactic conditions, which characterize executable specifications.

- **Sorts:** All sorts have to be defined with the `data` construct.

- **Modules:** Since Gofer does not have a module concept, executable specifications cannot include specification building operators like the `enriches` statement. Executable specifications have to be flat[1].
- **Functions:** Since Gofer is a functional language with pattern matching, all functions may be defined by pattern matching equations. The patterns must be linear and must be built of constructor functions. They must not overlap because otherwise the order of the axioms would determine the program behaviour. The right hand side of the equations may only contain applications of functions which are defined in the same way.
- **Classes:** The main reason for using Gofer was its support of type classes. Therefore executable specifications may include type classes, but all member functions of these classes have to be instantiated correctly by explicitly defined executable functions (see Section 5.2 for details).
- **Strictness:** Gofer is a lazy language but also supports strict functions[2]. We require all functions to be declared `strict`.

5.2 Executable Specification

The design specification of Section 4 describes all algorithms implementing the requirement specification but is not executable. Some simple development steps lead to an executable specification which is quite similar to the design specification. In this section we give some parts of the executable specification. The proof obligations which ensure the correctness of the executable specification with respect to the design specification are given in Section 5.3.

Gofer cannot support SPECTRUM's logical operations, since they are defined as parallel operations (see [BFG+93b]). Therefore SPECTRUM's logical operations are not executable. To yield an executable specification we have to avoid the use of these functions when defining our executable functions `insert`, `delete`, etc[3]. To do this we introduce a specification `SEQBool` which provides sequential boolean functions which are executable and replace SPECTRUM's boolean functions by them. The correctness of this step has to be proved.

```
SEQBool = { -- Specification of sequential boolean functions
   .&&. : Bool × Bool → Bool            prio 3:right;
   .||. : Bool × Bool → Bool            prio 2:right;

   axioms ∀⊥ x,y : Bool in
```

[1] We have some `enriches`-statements in the text since it is shorter than including the basic specifications into the text.

[2] Functions may by executed strictly, but constructor functions not. We expect this to change in further versions of Gofer.

[3] Thus we do not implement SPECTRUM's logical operations.

```
      (x && y) = if x then y else false endif;
      (x || y) = if x then true else y endif;
   endaxioms;
}
```

As mentioned in 4.5 in Gofer every member function for a class has to be instantiated by a concrete function. Modelling these instantiations in SPECTRUM gives us the possibility to prove the correctness of the instantiation which is not checked by Gofer. To demonstrate this we focus on the specification of trees. `Tree` α belongs to the class `EQ`, provided that the parameter sort α belongs to this class too, which is denoted in SPECTRUM by `Tree::EQ(EQ)`. This means that we can use the weak equality defined on the Sort α to define an instance function on `Tree` α (called `eqtree`).

```
Bintree_ex0 = {enriches SEQBool + Nat_ex;
-- executable specification of trees
   strict;
   data Tree α = emptytree
               | mktree(!node:α, !left:Tree α, !right:Tree α);
   eqtree: α::EQ ⇒ Tree α × Tree α → Bool;

   axioms α::EQ ⇒ ∀ a,n,x,y,n:α, s,t,u,v:Tree α in
       {eqtree1}   eqtree(emptytree,emptytree) = true;
       {eqtree2}   eqtree(emptytree,mktree(x,s,t)) = false;
       {eqtree3}   eqtree(mktree(x,s,t),emptytree) = false;
       {eqtree4}   eqtree(mktree(x,s,t),mktree(y,u,v)) =
                        ((x==y) && eqtree(s,u) && eqtree(t,v));
   endaxioms;
}

Bintree_ex = { enriches Bintree_ex0;
   strict;
   Tree :: (EQ)EQ;
   height : Tree α → Nat;
   .isin.: α::EQ ⇒ α × Tree α → Bool    prio 6;

   axioms α::EQ ⇒ ∀ a,n,n:α, l,r,s,t:Tree α in
       {inst}      (s==t) = eqtree(s,t);  -- Instantiating the weak equality
       {isin1}     a isin emptytree = false;
       {isin2}     a isin mktree(n,l,r) = ((a == n) || (a isin l) || (a isin r));
   endaxioms;

   axioms ∀ n:α, l,r:Tree α in
       {height1}   height(emptytree) = 0;
       {height2}   height(mktree(n,l,r)) = succ(max(height(l),height(r)));
```

```
   endaxioms;
}
```

In the design specification there were overlapping patterns in the definition of the function `delete` (`de_del2`–`3`). We replace these definitions by adding new, non-overlapping axioms for `delete`

```
{ex_del2}       delete(x,mktree(n,emptytree,emptytree)) = ...
{ex_del3}       delete(x,mktree(n,mktree(n1,l1,r1),emptytree)) = ...
{ex_del4}       delete(x,mktree(n,emptytree,mktree(n2,l2,r2))) = ...
```

We give the proof obligations which ensure the correctness of this step in the next section.

5.3 Proof Obligations

To ensure the correctness of the resulting program we have to show that the executable specification is an implementation of the requirement specification (assuming the correctness of the translation). Because of the transitivity of our implementation relation it suffices to show that the executable specification is an implementation of the design specification. In this section, we give the proof obligations resulting from the transformations carried out in Section 5.2 which guarantee that the executable specification is an implementation of the design specification.

Boolean Functions: The design specification has axioms which use SPECTRUM's logical connectives. Since we cannot implement these operations we replaced them in all axioms where they occur by those defined in `SEQBool`. We have to prove this replacement for every function which uses SPECTRUM's logical connections. In the example of `isbal` we get the following axioms:

```
axioms ∀ n:α, l,r:Tree α in
      isbal emptytree = true;
      isbal (mktree(n,l,r)) = ( (isbal l) && (isbal r) &&
         (  (height l == height r)
         || (height l == succ(height r))
         || (succ(height l) == height r) ) );
endaxioms;
```

With the help of these axioms we have to prove the axioms of the design specification[4]:

[4]The sequential Boolean functions of `SEQBool` coincide with the built-in parallel Boolean functions on defined values. Therefore the main part of the proof is to show that the arguments are defined at each application.

```
axioms ∀ n:α, l,r:Tree α in
      isbal(emptytree);
      isbal(mktree(n,l,r)) = ( (isbal l) ∧ (isbal r) ∧
         ( (height l  == height r)
         ∨ (height l == succ(height r))
         ∨ (succ(height l) == height r) ) );
endaxioms;
```

The same replacement has to be done in the axioms of the function `isord` and in some axioms of the primitive specification `Bintree_des`.

Correct Instances of Classes: The specification `Bintree_ex` contains the class information `Tree::(EQ)EQ;` which means that `Tree` α is a sort in class `EQ` if the parameter sort α is a sort in the sort class `EQ`. But this means, due to the specification of the class `EQ`, that there is a (member) function .==. for weak equality on type `Tree` α. With the instantiation axiom {`inst`} we declare the function `eqtree` to be the instance of the member function. This in turn means that the axioms of the member function .==. carry over to the function `eqtree` which can introduce an inconsistency if `eqtree` does not behave like a weak equality. Therefore we have to show that the instance `eqtree` of the member function .==. fulfills the axioms of the class `EQ` *before* we add the class information `Tree::(EQ)EQ;` and the instantiation axiom {`inst`} in the specification `Bintree_ex`.

Since the function .==. is defined in SPECTRUM's predefined specification in the following way

```
axioms α::EQ ⇒ ∀ a,b: α in
   {weak_eq}   (a==b) = (a=b);
endaxioms;
```

we have to show on the basis of the specification `Bintree_ex0` that

```
axioms α::EQ ⇒ ∀ a,b: Tree α in
     eqtree(a,b) = (a=b);
endaxioms;
```

To prove this we may use the axioms `eqtree1`–`4` which base on the .==. function available on all sorts of the class `EQ`.

The instantiations of the other classes have to be proved in the same way. Doing this we have to respect the subclass hierarchy. This means for example instantiating a sort in the class `TO` which is a subclass of `EQ` we have to prove the correct instantiations for both classes.

Overlapping Patterns: An executable specification must not contain any overlapping patterns. Simple transformations lead to the desired specification. Now we give the proof obligations which ensure that these transformations yield a specification which is an implementation.

For the replacement of the axioms `de_del2−3` of the design specification by the new axioms `ex_del2−4` we have to prove that the axioms with the overlapping patterns (`de_del2`, `de_del3`) are theorems of the executable specification.

There are no further proof obligations to this point since `delete` is the only function which used overlapping patterns in its axioms of the design specification.

Chapter 6

Transforming into a Functional Programming Language

In [Sud93] an ML program was developed out of the requirement specification. Since ML does not support type classes a longer development was necessary until the specification was in the executable part of SPECTRUM which corresponds to ML. In this paper we chose the functional programming language Gofer, since it supports the use of type classes and therefore the development process is shorter. This allowed us to present more formal details of the deductive software development. Other functional languages like OPAL would result in other developments.

Some trivial syntactic transformations of the executable specification yield an executable Gofer program. Just to give an impression we list the Gofer part for `Bintree`:

```
--************************************************************************
-- PART: Bintree

-- data type Bintree
data Tree a = Emptytree | Mktree(a,(Tree a),(Tree a))

-- selectors
------------
node (Mktree(x,l,r)) = x
left (Mktree(x,l,r)) = l
right (Mktree(x,l,r)) = r

-- function for class EQ
------------------------
eqtree Emptytree Emptytree = True
eqtree Emptytree (Mktree(x,s,t)) = False
eqtree (Mktree(x,s,t)) Emptytree = False
eqtree (Mktree(x,s,t)) (Mktree(y,u,v)) = x==y && (eqtree s u) && (eqtree t v)
```

```
-- instance of EQ
-----------------
instance Eq a => Eq (Tree a) where x==y = eqtree x y

-- function height
-------------------
height Emptytree = Zero
height (Mktree(n,l,r)) = Succ(maxnat(height l, height r))

-- function isin
-----------------
a `isin` Emptytree = False
a `isin` (Mktree(n,l,r)) = a == n || a `isin` l || a `isin` r
```

Chapter 7

Conclusion

This paper has (with a small example) led through the different steps of software development with SPECTRUM. Even though the example of AVL trees is a very simple one because it does not require data refinement and thus uses a simple notion of implementation, it is in our opinion well suited to motivate the necessity for the different development stages.

A comparison of the requirement specification of AVL trees in Section 3 and the corresponding design specification in Section 4.3 makes the importance of a good requirement specification clear. The requirement specification is quite short and easily understandable for everyone who is able to read first-order logic formulae. It states in a very compact form which properties are typical (and thus invariant) for AVL trees. Even an experienced specifier, however, can hardly deduce the typical features of AVL trees from the lengthy and complicated design specification. The notion of an executable specification, on the other hand, is introduced to free the design specification of any technical detail of a specific implementation language.

Through all these stages of development, SPECTRUM turned out to be a suitable language. While the expressiveness of its full first-order logic allowed us to write very abstract requirement specifications which are near to the informal description of the problem, SPECTRUM's built-in functional language features like the polymorphic type system were valuable for the later development steps.

To summarize, we advocate an approach in which the software development process is carried out as far as possible in the specification language (in our case SPECTRUM), whereas the transition to the implementation language (Gofer) is done as a very last simple transformation step. This has the advantage that we can use SPECTRUM's logical calculus to formally justify our steps throughout the whole development. In our example a four step development (requirement, design, executable specification and functional program) has proven to be suitable. For bigger developments, intermediate steps may become necessary which make the borders between those phases more floating.

Bibliography

[AVL62] G. M. Adelson-Velskii and Y. M. Landis. An algorithm for the organization of information. *Doklady Akademia Nauk SSSR*, (146):263–266, 1962. English translation: Soviet Math. 3, 1259–1263.

[BFG+93a] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Secification Language SPECTRUM. An Informal Introduction. Version 1.0. Part I. Technical Report TUM-I9311, Technische Universität München. Institut für Informatik, May 1993.

[BFG+93b] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Secification Language SPECTRUM. An Informal Introduction. Version 1.0. Part II. Technical Report TUM-I9312, Technische Universität München. Institut für Informatik, May 1993.

[HJW92] P. Hudak, S. Peyton Jones, and P. Wadler, editors. *Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2).* ACM SIGPLAN Notices, May 1992.

[Jon93] M. P. Jones. *An Introduction to Gofer*, August 1993.

[Pau93] L. C. Paulson. Introduction to Isabelle. Technical Report 280, University of Cambridge, Computer Laboratory, 1993.

[Pus94] C. Pusch. Verifikation einer Entwicklung von AVL-Bäumen in Isabelle. Master's thesis, Technische Universität München, 1994.

[Sch90] G. Schellhorn. Beispiele zur Verifikation von Moduln in Dynamischer Logik, 1990. Studienarbeit, Uni Karlsruhe.

[Sud93] C. Sudergat. Entwicklung von AVL-Bäumen in SPECTRUM. Fortgeschrittenenpraktikum, Technische Universität München, 1993.

[Wir76] N. Wirth. *Algorithms + Data Structures = Programs.* Prentice Hall, 1976.