Deriving an Applicative Heapsort Algorithm

Walter N. Guttmann[*]
University of Ulm

December 10, 2002

Abstract

"The development of sorting routines is a well trodden path." [DF88, p. 66]
We proceed by program transformation to derive a version of the Heapsort algorithm from a non-deterministic specification. The object language is Haskell and, therefore, the style of the resulting Heapsort program is necessarily applicative. Our development is supported by the program transformation system *Ultra*. Although a valuable tool, several shortcomings of *Ultra* are identified.

0 Overview

We will first describe the context in which the development is carried out. In Section 1 we will develop the initial specification that underlies the whole transformation task. Nothing more (and presumably not much less) has to be postulated to derive Heapsort. Section 2 points out good reasons for accepting these requirements. In Section 3 we will describe the derivation of Heapsort. Section 4 concludes with remarks on the adequacy of *Ultra*. The Appendix contains specifications, proofs, and the source code of the derivation (except for the derivation protocols).

0.1 Scope

The construction of programs from formal problem specifications by stepwise program transformations is one method to guarantee the correctness of solutions. The technique, transformational programming, is thoroughly described in [Par90].

The development of sorting algorithms in particular has a long tradition. The most recent update on the interrelations of various sorting algorithms is given by [ML97] that also includes references to significant work in their transformational synthesis. To our knowledge, the paper at hand contains the first derivation of Heapsort that persistently proceeds in a transformational manner.

A related treatment of sorting is found in [Par83] that, however, focuses on the intertwined development of control and data structure and also leads to a different sorting algorithm, namely sorting by replacement selection. In our approach, the fixed choice of trees as the intermediate data structure considerably simplifies the derivation. We are able to develop the algorithm in an even more constructive way. On the other hand, we abstain from the transition to the imperative paradigm and further optimisations.

[*]http://www.informatik.uni-ulm.de/pm/mitarbeiter/walter/

Our derivation is carried out using the program transformation system *Ultra* that is described in [Gut00]. Starting with descriptive specifications, the goal is to derive efficient operational programs. To this end the semi-automatic system *Ultra* supports the unfold-fold methodology. Programs are manipulated interactively by application of correctness preserving transformation rules.

0.2 Notational conventions

Generally we use normal text to display source code with keywords in bold typeface. In some cases we use mathematical symbols that, however, can be translated to ASCII in a straight-forward manner:

for this symbol	write this ASCII text	
λ	`\`	
$\rightarrow$	`->`	
$\wedge$	`&&`	
$\vee$	`||`	
$\leq$	`<=`	
$\Longleftrightarrow$	`<=>`	
$=$	`==`	unless = is definitional
some x : p x	`some (\x -> p x)`	for arbitrary predicate p

0.3 General remarks

By "algebraic data type" we mean a data type declared with the Haskell "data" construct. In [Par90] the phrase "mode" is used for this. There, the phrase "algebraic type" is used for what we call "abstract data type", since algebras are their models.

Applying the unfold-fold transformation methodology guarantees only partial correctness. Therefore, termination must be proved separately, a task for which *Ultra* is not suited. Anyway, a proof that the resulting Heapsort algorithm has $O(n \log n)$ time complexity must also be given separately and that implies termination.

We assume that all transformation rules are applied to terms that are well-defined. This requirement is not explicitly mentioned in the applicability conditions of the transformation rules because *Ultra* does not support reasoning about definedness properties. Note that several transformation rules are not valid if definedness is not presumed. However, all functions in the specification we are about to give and from which Heapsort is derived are well-defined.

1 Problem specification

As specification language we use the functional language Haskell. To support descriptive specifications we have extended the object language by non-deterministic expressions. On the other hand, we have restricted the object language by not supporting some of the advanced features of Haskell such as type classes. This is not a problem for our sorting algorithm if we assume that there is a linear ordering on the type of the elements that have to be sorted.

The semantics of the object language is described in [Sch98] and is strongly based on [Pat92]. According to the classification of [SS92] it has a non-strict, singular, angelic non-determinism. We are aware of the subtle problems that arise with non-determinism or call-by-need in program transformation. These raise no difficulties, however, for the derivation described here, yet they need to be investigated further.

We start with the following (traditionally) correct description of sorting:

```
sort                ::  [a] → [a]
sort xs             =   some ys : issorted ys ∧
                               elementsof ys ‘equals‘ elementsof xs

issorted            ::  [a] → Bool
issorted [ ]        =   True
issorted (x:xs)     =   x = minimum (x:xs) ∧ issorted xs

minimum             ::  [a] → a
minimum [x]         =   x
minimum (x:y:zs)    =   x ‘min‘ minimum (y:zs)

min                 ::  a → a → a
min x y             =   if x ≤ y then x else y

elementsof          ::  [a] → Bag a
elementsof [ ]      =   emptybag
elementsof (x:xs)   =   nonemptybag x (elementsof xs)
```

A few remarks are appropriate here. The some-expression in the function sort selects (non-deterministically) a list ys that satisfies the given predicate.

Although we spelled out an explicit definition for minimum and min, we will not need the defining equations during the constructive part of the derivation. They are needed, however, for the proof of consistency of the theory we are about to present.

The assumption that the elements that have to be sorted are linearly ordered becomes manifest in the operation ≤ in the definition of min. In Haskell, this operation is provided in the Ord type class so that strictly speaking it is required to add the constraint “Ord a” to the type declarations. We will omit this constraint in the following since our transformation system makes no use of it anyway. Alternatively, we could also abandon polymorphism and instantiate the declarations by a concrete element type such as Int, that has a ≤ operation due to its membership in the Ord type class.

The requirement that the sorted list is a permutation of the input list led to the definition of elementsof. It constructs a bag (or multiset) of the elements of its parameter list. Moreover, we use the equality (on bags) equals. Within the traditional specification, the type Bag is meant to be an abstract data type. Our transformation system *Ultra* has no means to define and manipulate abstract data types directly. We will emulate abstract data types by providing an algebraic data type, primitive functions (i.e., functions without defining equations), and transformation rules as axioms. The definition of elementsof uses this algebraic data type (called Bag) and its two generator functions, namely emptybag and nonemptybag.

The presumed specification of the abstract data type Bag is given in Appendix A. It is translated into the *Ultra* theory file “bag.thy” that is given in Appendix D.4. We make use of *Ultra*’s facility to define algebraic properties instead of providing transformation rules for the associativity, commutativity, reflexivity, and transitivity axioms.

The theory file “bag.thy” contains almost the whole specification of bags. It is based on several functions that are provided through the Haskell prelude. Since one of them, namely minimum, is not available in *Ultra*, we have declared it in an extra file "general.thy" shown in Appendix D.2. That file also contains some of the axioms from the theories the specification of bags is based on. Additionally, it contains several other transformation rules that state properties of the object language and will be discussed in Section 2.1.

To complete the problem specification, sort and issorted are defined in the theory file "sort.thy" (see Appendix D.9). The following slightly adapted quotation from [Par83] tells us, how to proceed. "The type Bag used so far (together with the minbag function) is in principle nothing but a priority queue. This observation leads to the idea (Eureka!) of representing Bag by a certain kind of binary trees." Although the idea of using heaps is essential, we need not make but a few assumptions for their specification.

Our specification of heaps is contained in the theory file "tree.thy" given in Appendix D.6. It declares an algebraic data type Tree that is used to implement bags and defines the appropriate abstraction function. A few axioms for heaps are stated that will be discussed in Section 2.3.

This concludes the specification of the starting point of our derivation. The current status can be summarised informally: if you buy what we said up to now, the rest follows semi-automatically. In order to encourage you, we will now argue for the adequacy of our specification.

2 Adequacy of the specification

Let us partition the specification into two parts. The first part contains functions and rules that are universally applicable, i.e., they are directly derivable from the object language. The second part consists of the domain-specific definitions. The basic assumption we have to make is that we use "some kind of" bags to define the notion of permutation and "some kind of" trees to implement these bags. Let us therefore split the second part into a theory of bags and a theory of trees. We will now deal with each of these three parts in turn.

2.1 Theory of general definitions

The theory file "general.thy" that is given in Appendix D.2 contains definitions that are based on the object language. The only function defined is minimum, and the reason for its occurrence is that it is not available through the standard prelude of *Ultra* (it is provided in the standard prelude of Haskell, yet). It follows by structural induction that it indeed calculates the minimum element of a list.

Several rules in "general.thy", namely reflexivity, transitivity, commutativity, left_neutral, and some_simplification, have applicability conditions (these are the only rules with applicability conditions in our specification). Such conditions need to be resolved, and *Ultra* can do this only if matching clauses are defined (see the end of Section D.2). From the semantics of the object language it follows that $\wedge$ is associative, commutative, and has True as a left neutral element and that $=$ is commutative. Therefore, commutativity and left_neutral describe sound transformations (for these instances). Further instances of these two rules and the other rules with applicability conditions are discussed in Section 2.2. By the way, the reason for explicitly stating a transformation rule for commutativity is an insufficiency of *Ultra*: the built-in rules only apply to operations that are both associative and commutative.

Three more rules, namely constant_lift_exists, some_split, and choice_and_quantification, are taken from [Man74, Par90, Sch98] as indicated in Section D.2. The soundness of these rules follows from the semantics of the object language, e.g., [Sch98, Appendix D, p. 113 ff] contains a proof of (a generalisation of) some_split. Note that all these rules state properties of non-deterministic expressions.

The correctness of the rule import_truth is readily verified by an inspection of the possible cases of the boolean parameter x. In the rule some_eliminate_assertion, the boolean parameter q must be true, since all terms are well-defined, hence it can be eliminated. Finally, for the same reason, in the rule some_simplification y must satisfy the predicate p. Since r is reflexive, we conclude that y is a solution for the some-expression. Note that the latter two rules use the descendance relation between the input and output program schemes instead of the equivalence relation (these are the only such rules in our specification).

2.2 Theory of bags

As we have already elaborated, the theory file "bag.thy" shown in Appendix D.4 has to be understood as a specification of an abstract data type. In *Ultra* this is ensured by declaring the constructor functions primitive, so that they cannot be unfolded and only structural equivalence is assumed. This way, the declared algebraic data type Bag is the term algebra model of the abstract data type Bag. There is an alternative way to model an abstract data type, namely by declaring it as the sum of its constructors. However, then a case-introduction on terms of type Bag can be performed where all constructors appear, not just the two generators.

The abstract data type Bag is specified in Appendix A. There, we also show that the mathematical algebra of bags is a model of this specification and that all its models are behaviourally equivalent (see Theorem A.10). Therefore, every possible implementation of our specification behaves as we are used to it. We will not, however, derive a direct (term generated) implementation in Haskell, since this is not necessary for the Heapsort algorithm. It could be obtained, e.g., by the techniques of [Par90, Chapter 8.2].

2.3 Theory of trees

The theory file "tree.thy" (see Appendix D.6) contains function and rule definitions for binary, node-labelled trees that are used as the intermediate data structure for Heapsort. We will use the algebraic data type Tree to implement bags, and the appropriate abstraction function is included in the theory file. It collects all elements of the binary tree into a bag. Additionally this theory file contains the definition of the function heapinv that defines which trees satisfy the heap property.

The specification of trees is not part of the Heapsort problem specification, but rather our design decision. Therefore, it is sufficient to show the consistency of the specification and not necessary to argue that all its models are "classical" binary trees. Even so, by declaring Tree as an algebraic data type, we fixed the model to be (freely) term generated.

Let us discuss the three axioms for trees. The rule some_split_tree is a special case of the rule discussed and proven correct in [Sch98, Appendix D]. Appendix B (of this paper) contains the main results for the other two axioms.

The rule tree_exists states that for every bag there is a representing heap, i.e., that abstraction is surjective even if its domain is restricted to heaps. It will be needed during the derivation of Heapsort, e.g., when we want to remove from a bag its minimum element: b'=without(b,minbag(b)). The bag b is represented by a heap, say h, and the resulting bag should also be represented by (another) heap, say h'. We need to calculate h' from h, and to this end we devise a function removemin with h'=removemin(h). A direct specification is

equals(without(abstraction(h), minbag(abstraction(h))),
    abstraction(removemin(h))) ≡ true,

but the consistency of this is difficult to show due to the nested occurrence of removemin. If the surjectivity of abstraction is granted, however, we can define removemin(h) to be some heap that is mapped by abstraction to the resulting bag. The surjectivity of abstraction is proved in Theorem B.1.

Finally, the rule tree_minimum requires that the minimum element of a bag is at the root of its representing heap. In this rule (and in other rules, too) we had to restrict the possible values of the data type Tree to be heaps. The type system of the object language, however, does not allow us to declare constrained algebraic data types. We would like to solve this problem by introducing the appropriate applicability conditions into our transformation rules. While this is basically possible in *Ultra*, the transformation system cannot reasonably deal with such applicability conditions, i.e., resolve them. Therefore, we encode the heap property as a conjunctive assertion here and in other rules. The validity of the rule tree_minimum is proved in Theorem B.2.

Since all rules are proved and the defined functions are total, there is a model of our theory of trees.

2.4 Theory of sorting

Finally, let us remark that the theory file "sort.thy" presented in Appendix D.9 contains the definitions of sort and issorted as given in the problem specification.

3 Summary of the derivation

The development of Heapsort is centred around the introduction of certain trees, namely heaps, as intermediate data structures and can be separated into two parts. The first part deals with the composition of an intermediate heap from the input sequence. The second part deals with the decomposition of the intermediate heap into the sorted output sequence. We give an overview of the most important transformation rules and their informal meaning in the context of the derivation.

composition	rules
There is a heap for every bag,	tree_exists
in particular for the bag given by the input list.	introduce_composetree
This heap can be constructed recursively	composetree_recursive
using the auxiliary function insert,	
that can be formulated recursively, too.	insert_recursive

decomposition	rules
The input sequence can be sorted	decomposetree_recursive
by recursively removing the minimum elements.	
The minimum element of the list is	bag_minimum_list
the minimum element of the bag,	
and for the bag there is a heap,	(by tree_exists)
in which the minimum element is the root.	tree_minimum
Removing this root leaves a collection of elements,	introduce_removemin
for which there is a heap,	(by tree_exists)
that is recursively constructible.	removemin_recursive

We will elaborate the order in which the derivation of Heapsort proceeds below. A detailed list containing all derived rules and their dependance graph is presented in Appendix C. We discuss the important parts of the derivation in a top-down fashion and thereby show a few parts in full detail to provide an impression of the actual work. When explaining the individual steps we refer to the rules by the numbering given in Appendix C to illustrate the state of the derivation.

3.1 Main development

The top level development of Heapsort is carried out in the derivation of the transformation rule main_development. It starts with the descriptive specification of the function sort as given in Section 1. We mark the ►selected sub-terms◄ that serve as the target for each step.

```
        ►sort xs◄
⟺      ⦃unfold⦄
        some ys : issorted ys ∧ ►elementsof ys ‘equals‘ elementsof xs◄
⟺      ⦃1.1. apply introduce_composetree (see Section 3.2)⦄
        some ys : ►issorted ys ∧
                  elementsof ys ‘equals‘ abstraction (composetree xs)◄
⟺      ⦃apply left_neutral⦄
        some ys : ►True◄ ∧ issorted ys ∧
                  elementsof ys ‘equals‘ abstraction (composetree xs)
⟺      ⦃1.2. apply heap_composetree backwards⦄
        some ys : heapinv ►(composetree xs)◄ ∧ issorted ys ∧
                  elementsof ys ‘equals‘ abstraction ►(composetree xs)◄
⟺      ⦃λ-abstraction (with multiple selection)⦄
        ►(λt → some ys : heapinv t ∧ issorted ys ∧
             elementsof ys ‘equals‘ abstraction t)◄ (composetree xs)
⟺      ⦃1.3. define decomposetree (that is automatically folded)⦄
        ►decomposetree (composetree xs)◄
⟺      ⦃1.4. define heapsort (again, automatically folded)⦄
        heapsort xs
```

Some sorted list is wanted that, when viewed as a bag, has the same elements as the bag containing the elements of the input sequence xs. A heap is introduced (see Section 3.2) as the intermediate data structure to represent the latter bag (1.1., rule introduce_composetree, defining the function composetree). Given such a heap, the construction of the required sequence is named decomposetree (1.3.) and is still not operational. The function composition of composetree and decomposetree is named heapsort (1.4.) which finishes the main development. Recursive, i.e., operational versions of composetree (2., rule composetree_recursive) and decomposetree (4., rule decomposetree_recursive) will be derived in Sections 3.3 and 3.5.

3.2 Introduction of heaps

The first sub-development introduces heaps and is carried out in the derivation of the transformation rule introduce_composetree. Given an arbitrary bag and the knowledge that there exists a heap representing it, we may describe the bag as the abstraction of some heap whose abstraction is the bag. This is a variant of a more general rule, namely

$$a \iff f(\textbf{some } x : f(x) = a), \text{ provided that } \exists x : f(x) = a.$$

Hence, it is no surprise that its derivation (1.1.1., rule equals_abstr_some_heap_equals_abstr) does not refer to properties of bags. It refers to trees, however, to avoid the introduction of the applicability condition. More precisely, we have not derived the rule in its full generality but instantiated the function f. Also, the equivalence stated in the general rule had to be replaced by the equality on bags. This is because only the existence of a heap that represents a bag up to behavioural equivalence is granted. Indeed, a version of the existence property that demands structural equivalence is false.

The transformation rule equals_abstr_some_heap_equals_abstr is used in the derivation of introduce_composetree:

```
      ►b ‘equals‘ elementsof xs◄
⟺    ⦃apply left_neutral⦄
      ►True◄ ∧ b ‘equals‘ elementsof xs
⟺    ⦃1.1.1. apply equals_abstr_some_heap_equals_abstr backwards⦄
      ►elementsof xs ‘equals‘ abstraction (some t : heapinv t ∧
           elementsof xs ‘equals‘ abstraction t) ∧ b ‘equals‘ elementsof xs◄
⟺    ⦃1.1.2. apply commutativity_transitivity⦄
      ►elementsof xs ‘equals‘ abstraction (some t : heapinv t ∧
           elementsof xs ‘equals‘ abstraction t)◄ ∧ b ‘equals‘ abstraction
           (some t : heapinv t ∧ elementsof xs ‘equals‘ abstraction t)
⟺    ⦃1.1.3. apply equals_abstr_some_heap_equals_abstr⦄
      ►True ∧ b ‘equals‘ abstraction (some t : heapinv t ∧
           elementsof xs ‘equals‘ abstraction t)◄
⟺    ⦃simplify⦄
      b ‘equals‘ abstraction ►(some t : heapinv t ∧
           elementsof xs ‘equals‘ abstraction t)◄
⟺    ⦃1.1.4. define composetree⦄
      b ‘equals‘ abstraction (composetree xs)
```

We have introduced some heap whose abstraction is the given bag containing the elements of the input sequence:

some t : heapinv t ∧ elementsof xs ‘equals‘ abstraction t.

This is a descriptive specification of a function to construct such a heap and it is named composetree (1.1.4.).

3.3 Recursive composition of heaps

The second sub-development derives a recursive, i.e., operational version of composetree in the rule composetree_recursive. It proceeds according to the unfold-fold methodology. We start with the expression

composetree xs.

The parameter xs of composetree contains the elements of a sequence (in the outermost call: of the input sequence). By unfolding composetree and elementsof we get sufficient detail to distinguish two cases: the empty and the non-empty sequence. By rearrangements we are able to deal with both cases in turn:

```
case xs of
    [ ] → some t : heapinv t ∧ emptybag ‘equals‘ abstraction t
    y:ys → some t : heapinv t ∧
                nonemptybag y (elementsof ys) ‘equals‘ abstraction t
```

In the first case we have to find some heap whose abstraction is the empty bag. The rule some_heap_equals_empty_abstr (2.1.) derives by case-inspection that there is but one such heap, namely the empty heap. Therefore, the non-determinism is spurious and can be solved. To illustrate the automation capabilities of *Ultra* we present the derivation of some_heap_equals_empty_abstr in full detail:

```
      some t : heapinv t ∧ emptybag ‘equals‘ ▸abstraction t◂
⟺    ⦃unfold⦄
      some t : ▸heapinv t ∧ emptybag ‘equals‘ case t of
          Leaf → emptybag
          Node l e r → abstraction l ‘unionbag‘ singletonbag e
                       ‘unionbag‘ abstraction r◂
⟺    ⦃simplify⦄
      some t : case t of
          Leaf → heapinv Leaf ∧ ▸emptybag ‘equals‘ emptybag◂
          Node l e r → heapinv (Node l e r) ∧ emptybag ‘equals‘
              abstraction l ‘unionbag‘ singletonbag e ‘unionbag‘
              abstraction r
⟺    ⦃apply reflexivity⦄
      some t : case t of
          Leaf → heapinv Leaf ∧ True
          Node l e r → heapinv (Node l e r) ∧ ▸emptybag ‘equals‘
              abstraction l ‘unionbag‘ singletonbag e ‘unionbag‘
              abstraction r◂
⟺    ⦃apply catalogue bag-derived.thy⦄
      ▸some t : case t of
          Leaf → heapinv Leaf ∧ True
          Node l e r → heapinv (Node l e r) ∧ False◂
⟺    ⦃solve⦄
      Leaf
```

In the second case we have to find some heap whose abstraction is a non-empty bag that consists of the first element of the sequence and the bag containing the remaining elements.

For the latter bag there exists a representing heap and from the first sub-development in Section 3.2 we already know how to construct it: with the function composetree (2.2.). It remains to find some heap whose abstraction consists of an element y and the abstraction of a heap s:

```
some t : heapinv s ∧ heapinv t ∧
    nonemptybag y (abstraction s) ‘equals‘ abstraction t.
```

This is a descriptive specification of a function to construct such a heap and it is named insert (2.4.) because its effect is to insert an element into a heap. This completes the second sub-development:

```
case xs of
    [ ] → Leaf
    y:ys → insert y (composetree ys)
```

A recursive, i.e., operational version of insert (3., rule insert_recursive) will be derived in Section 3.4.

The resulting rule describes a recursive definition of composetree, yet we cannot use *Ultra* to translate it automatically. Therefore, we have to manually extract the recursive function and it is shown in Appendix D.12.

3.4 Recursive insertion into heaps

Before we embark on the decomposition of heaps, we derive a recursive, i.e., operational version of insert in the rule insert_recursive. The third sub-development proceeds according to the unfold-fold methodology, too. We start with the expression

```
insert x t.
```

The parameter t of insert contains the elements of a heap (in the outermost call: of the empty heap). By unfolding insert and abstraction we get sufficient detail to distinguish two cases: the empty and the non-empty heap. By rearrangements we are able to deal with both cases in turn:

```
case t of
    Leaf → some s : heapinv Leaf ∧ heapinv s ∧
                nonemptybag x emptybag ‘equals‘ abstraction s
    Node l e r → some s : heapinv (Node l e r) ∧ heapinv s ∧
                nonemptybag x (abstraction l ‘unionbag‘ singletonbag e
                    ‘unionbag‘ abstraction r) ‘equals‘ abstraction s
```

In the first case we have to find some heap whose abstraction is a singleton bag. The rule some_heap_equals_nonempty_abstr (3.1.) derives by case-inspection that there is but one such heap, namely the heap with one node whose value is the element that is to be inserted. Therefore, the non-determinism is spurious and can be solved.

In the second case we have to find some heap whose abstraction contains the element x that is to be inserted, the root element e of the tree t, and the abstractions of both sub-trees l and r of t. If we use set-notation for our bags and their operators, we may write this as follows:

```
some t : heapinv t ∧ abstraction t ‘equals‘
    {x} ∪ {e} ∪ abstraction l ∪ abstraction r.
```

Since the resulting heap must have the minimum element at its root, and e is the minimum element of the heap t, we distinguish two cases: either x≤e or x>e. This case introduction is formally performed by using the rule heap_insert from the theory file “tree-added.thy” that is proven correct in Appendix B. In the first case (3.4., rule some_heap_insert_1) we rearrange the abstractions using associativity and commutativity of unionbag as follows:

```
some t : heapinv t ∧ abstraction t ‘equals‘
    ({e} ∪ abstraction r) ∪ {x} ∪ abstraction l.
```

For the first bag ({e} ∪ abstraction(l)) there exists a representing heap (3.4.1.1.1.) and from the second sub-development in Section 3.3 we already know how to construct it: with the function insert (3.4.1.1., rule heap_equals_nonempty_abstr_abstr_insert). After this, we can fold abstraction and get the resulting representing heap. In the second case (3.5., rule some_heap_insert_2) we rearrange as follows:

```
some t : heapinv t ∧ abstraction t ‘equals‘
    ({x} ∪ abstraction r) ∪ {e} ∪ abstraction l.
```

We proceed through the same steps as before to finish this case, too. This completes the derivation of a recursive version of insert:

```
case t of
    Leaf → Node Leaf x Leaf
    Node l e r → if e ≤ x then Node (insert x r) e l
                              else Node (insert e r) x l
```

Again, we have to manually extract the recursive function that is shown in Appendix D.12.

Finally, let us remark that there are other possible ways to combine x, e, l, and r to a heap (maybe even involving further information such as the sizes or depths of the sub-trees). The variants we have chosen lead to the construction of “Braun trees” that are balanced and guarantee access with $O(\log n)$ time complexity.

3.5 Recursive decomposition of heaps

The fourth sub-development derives a recursive, i.e., operational version of decomposetree in the rule decomposetree_recursive. Again, it proceeds according to the unfold-fold methodology. We start with the expression

```
decomposetree t.
```

It is not necessary to make any assumptions on what elements the input heap t consists of (e.g., that it was constructed by composetree). We can distinguish two cases: the empty and the non-empty input heap, and we deal with them in turn:

```
case t of
    Leaf → some xs : heapinv Leaf ∧ issorted xs ∧
                elementsof xs ‘equals‘ abstraction Leaf
    Node l e r → some xs : heapinv (Node l e r) ∧ issorted xs ∧
                elementsof xs ‘equals‘ abstraction (Node l e r)
```

In the first case we have to find some sorted list whose abstraction is the abstraction of the empty heap, i.e., the empty bag. The rule some_sorted_equals_elements_empty (4.1.) derives by case-inspection that there is but one such list, namely the empty list. Therefore, the non-determinism is spurious and can be solved.

In the second case we have to find some sorted list whose abstraction is the abstraction of a non-empty heap, i.e., a non-empty bag:

some xs : issorted xs $\wedge$ elementsof xs 'equals' abstraction (Node l e r).

By case-inspection we are able to rule out the empty list as a candidate. From the requirement that the non-empty list must be sorted, we can conclude that its head must be its minimum element and, therefore, the minimum element of its abstraction and, hence, the minimum element of the abstraction of the input heap, i.e., its root. We also conclude that the tail of the non-empty list must be sorted.

From the equality of the two bags it follows that the abstraction of the tail is the abstraction of the input heap without its minimum element. We already know from the first sub-development in Section 3.2 that for the latter bag there exists a representing heap (4.4.1.2.). This gives us a descriptive specification to construct such a heap and it is named removemin (4.4.1.5.) because its effect is to remove the minimum element of a heap s:

```
some t : heapinv s ∧ heapinv t ∧
    (abstraction s 'without' minbag (abstraction s)) 'equals' abstraction t.
```

We now have separate descriptions for how to construct the head and the tail of the sorted list:

```
some (y:ys) : y = e ∧ heapinv (Node l e r) ∧ issorted ys ∧
    elementsof ys 'equals' abstraction (removemin (Node l e r)).
```

For the tail we have to find some sorted list whose abstraction is the abstraction of a heap and from the main development in Section 3.1 we already know how to construct it: with the function decomposetree. This completes the fourth sub-development:

```
case t of
    Leaf → [ ]
    Node l e r → e : decomposetree (removemin (Node l e r))
```

A recursive, i.e., operational version of the function removemin (5., rule removemin_recursive) will be derived in Section 3.6. Again, we have to manually extract the recursive function that is shown in Appendix D.12.

3.6 Recursive removal from heaps

We derive a recursive, i.e., operational version of removemin in the rule removemin_recursive. The fifth sub-development proceeds according to the unfold-fold methodology, too. We start with the expression

removemin t.

Note that from the context we know that removemin is never called with the empty heap as its parameter t. We can distinguish two cases: the empty and the non-empty input heap, and we deal with them in turn:

```
case t of
    Leaf → some s : heapinv Leaf ∧ heapinv s ∧
            (abstraction Leaf ‘without‘
                minbag (abstraction Leaf)) ‘equals‘ abstraction s
    Node l e r → some s : heapinv (Node l e r) ∧ heapinv s ∧
            (abstraction (Node l e r) ‘without‘
                minbag (abstraction (Node l e r))) ‘equals‘ abstraction s
```

In the first case we have to find some heap whose abstraction is the abstraction of the empty heap without its minimum element. Since this case is never invoked in our algorithm it is no surprise that, using a strict semantics, the value of removemin is undefined here because the empty heap has no minimum element. Even so, using our lazy semantics, we can conclude that the abstraction of the empty heap, i.e., the empty bag, without any element is still the empty bag. Therefore, the empty heap satisfies the descriptive specification in this case (5.1.).

In the second case we have to find some heap whose abstraction is the abstraction of a non-empty heap without its minimum element. Since the minimum element of the heap is its root, removing it leaves us with the union of the abstractions of both sub-trees. We will distinguish several cases.

If the left sub-tree is empty, the union is the abstraction of the right sub-tree and, therefore, the right sub-tree solves the specification (5.3.1., rule some_heap_equals_union_empty_abstr). If the left sub-tree is non-empty and the right sub-tree is empty, the union is the abstraction of the left sub-tree and, therefore, the left sub-tree solves the specification (5.3.2., rule some_heap_equals_union_abstr_empty). Otherwise both sub-trees are non-empty and using set-notation for our bags we may write this as follows:

```
some t : heapinv t ∧ abstraction t ‘equals‘
    abstraction (Node l’ e’ r’) ∪ abstraction (Node l” e” r”).
```

Since the resulting heap must have the minimum element at its root, and e’ and e” are the minimum elements of the left and right sub-trees, we distinguish two cases: either e’≤e” or e’>e”. This case introduction is formally performed by using the rule heap_removemin_case from the theory file “sort-added.thy” that is proven correct in Appendix B. In the first case (5.3.4., rule some_heap_remove_1) we rearrange the abstractions on the right hand side using associativity and commutativity of unionbag as follows:

```
some t : heapinv t ∧ abstraction t ‘equals‘
    (abstraction l’ ∪ abstraction r’) ∪ {e’} ∪ abstraction (Node l” e” r”).
```

For the first bag (abstraction(l’) ∪ abstraction(r’)) there exists a representing tree (5.3.4.1.1.2.) and from the fourth sub-development in Section 3.5 we already know how to construct it: with the function removemin (5.3.4.1.1., rule heap_equals_union_abstr_abstr_abstr_remove). After this, we can fold abstraction and get the resulting representing heap. In the second case (5.3.5., rule some_heap_remove_2) we rearrange as follows:

```
some t : heapinv t ∧ abstraction t ‘equals‘
    abstraction (Node l’ e’ r’) ∪ {e”} ∪ (abstraction l” ∪ abstraction r”).
```

We proceed through the same steps as before to finish this case, too. This completes the derivation of a recursive version of removemin:

```
case t of
    Leaf → Leaf
    Node l e r → case l of
        Leaf → r
        Node l' e' r' → case r of
            Leaf → l
            Node l" e" r" → if e' ≤ e" then Node (removemin l) e' r
                                       else Node l e" (removemin r)
```

Again, we have to manually extract the recursive function that is shown in Appendix D.12.

4 Adequacy of *Ultra*

The derivation of Heapsort shows the strength of the program transformation system *Ultra* but also points out several of its shortcomings. We first discuss the support of the transformation calculus, then we have a look at the object language, thereafter we continue with system issues, and finally we conclude with general remarks.

4.1 Support of the transformation calculus

Ultra implements most of the transformation calculus as described by [Pep87] and elaborated in [Par90]. The unfold-fold methodology is completely supported and complemented with powerful tactics.

The most important drawback is the lack of adequate support for transformation rules with applicability conditions. At the moment only conditions stating simple algebraic properties can be reasonably used, however, these are well supported. In our derivation the need for (inductively defined) invariants for data structures and simple arithmetic assertions arose. A possible way to resolve such conditions is provided by theorem proving. The exact kind of interaction between transformation and proving needs to be investigated further.

Most of the facts that we had to prove manually (see Appendices A and B) could have been derived in *Ultra* if the system supported induction proofs. Although we have investigated on this topic, induction proofs are not integrated into *Ultra* yet.

4.2 Language support

The object language of *Ultra* is Haskell with additional declarative constructs but without some advanced concepts such as type classes. Quite contrary to the CIP system, there are no intentions to provide imperative features in our language. Therefore, we inherit most of the advantages and disadvantages of functional programming languages. The supported subset is adequate to be able to reason about small-sized programs or small-sized parts of large programs (observing that functional programs typically consist of many, but rather small functions). The added declarative constructs, especially the some-expressions, allow the concise formulation of non-deterministic specifications from which operational programs can be derived semi-automatically.

Even if no proper transformation support for some constructs is provided, *Ultra* should be able to process the complete Haskell language. Several attempts have been made by others to formalise the semantics of (parts of) Haskell, but still there is no standard formal semantics for the complete language. While this is not a problem for the transformational approach, it is not satisfactory to produce programs in *Ultra* that might have a different behaviour using a Haskell compiler. Therefore, the use of a variant of Haskell as object language should be questioned.

Abstract data types provided the vehicle for the specification of the components for the Heapsort derivation. There is no direct support for abstract data types in *Ultra*. Instead we had to use primitive functions and globally visible transformation rules as axioms.

One of the decisions that have to be made if abstract data types are introduced is the underlying logic. A proper balance between expressivity on the one hand and usability and decidability on the other hand must be found. The transition from specifications as abstract data types to implementations must be supported. This is simple for constructive specifications (e.g., elembag in BAG), but difficult for declarative ones (e.g., minbag or the associativity axiom of unionbag in BAG).

Finally, it is necessary to provide some kind of modularisation facilities to make a clean transition possible. For instance, the dependances between theory files must be stated explicitly. A key concept for the solution of this problem is termed "conservative extension". The literature, however, provides plenty of definitions and realisations of this concept and it remains unclear which variant to choose in the context of our transformational approach.

4.3 System issues

The graphical user interface of *Ultra* allows convenient access to transformation rules, functions, and the tasks of the unfold-fold methodology and supporting tactics. Context-sensitive editing and navigation facilities enable a comfortable derivation. The built-in simplifier, that can be called automatically after each transformation step, polishes the resulting expressions.

In some stages of the derivation of Heapsort the user has to resort to manually modifying theory files. Although it is not the primary concern of a transformation system to provide many kinds of editing possibilities, the important manipulations have to be integrated. For instance the derivation of structural recursive function definitions from (certain kinds of) specifications given as transformation rules was needed several times in Section 3.

Whenever a derivation has been completed, the only remaining artifacts are the resulting transformation rule and the derivation protocol. While the transformation rule can be used in another derivation, the protocol serves purely for documentation purposes. No arrangements have been made for accessing the protocol in any way from within *Ultra*. The protocol should capture the derivation process in a way that allows a replay and a refactoring of derivations. By replay we mean firstly that similar derivation steps can be performed on another term during another derivation, and secondly that the derivation itself is amenable to small changes. By refactoring we mean that it is possible to factor out parts of the derivation into auxiliary transformation rules and to rearrange parts of the derivation.

Although *Ultra* has several built-in tactics, the level of automation should be still enhanced. This could be supported by a proper use of all available context information. Higher-order pattern matching and the application of rules modulo associativity and commutativity would shorten derivations considerably, too. On the other hand, a large library of simple transformation rules that can be manually used to bend the expressions until they fit the user's needs is missing, too. For instance, while the system can automatically distribute function calls over conditionals, the user has to define transformation rules on his own for the backward direction.

4.4 Conclusion

The core of the specification of Heapsort (see Section 1) as well as the resulting program (see Appendix D.12) are not large. Even so, the derivation of quite a few transformation rules is necessary to provide the transition, as can be verified from Appendix E. We assume that a purely verificational approach to prove the correctness of a specific Heapsort implementation needs similar efforts. For a reasonable comparison, it must be noted that several of the transformation rules that we have derived (e.g., those about bags) might already be part of a library in other systems.

Once more we note that it is expensive (in terms of time and knowledge requirements) to develop correct programs. Besides systems where correctness must be established by all means, we see two other areas of application. First, the number of uses of a certain software piece may be large enough to legitimate high development costs, e.g., in software libraries, code generators, and within embedded systems. Second, a light-weight interface to any kind of formally based "correctness-improving" system might be integrated into conventional development environments.

A Specification of the abstract data type Bag

Informally, a bag is like a set with the difference that it may contain multiple occurrences of elements. We present a specification of the abstract data type Bag in the style of [Par90]. The following components are taken from that book:

- LINORD as required for exercise 3.3-5b, page 143
- BOOL as given on page 66
- MSEQU with the appropriate renamings from EESEQU and minel on pages 202-203
- ASSOC as given on page 77
- COMM with the law for commutativity (exercise 3.3-5f in some versions)
- EQUIV as given on page 76

This is the specification of the abstract data type Bag:

```
type BAG = (sort elem, funct (elem, elem) bool .≤., funct (elem, elem) bool .=.)
                bag, emptybag, nonemptybag, singletonbag, unionbag,
                without, elembag, equals, minbag, elementsof:
    include LINORD(elem, ≤, =),
    based on BOOL,
    based on (elemsequ, [ ], .:., minimum) from MSEQU(elem),
    sort bag,
    bag emptybag,
    funct (elem, bag) bag nonemptybag,
    funct (elem) bag singletonbag,
    funct (bag, bag) bag unionbag,
    funct (bag, elem) bag without,
    funct (elem, bag) bool elembag,
    funct (bag, bag) bool equals,
    funct (bag b: ¬equals(b, emptybag)) elem minbag,
    funct (elemsequ) bag elementsof,
    include ASSOC(bag, unionbag),
    include COMM(bag, unionbag),
    include EQUIV(bag, equals),
    laws elem e, e', bag b, b', b", elemsequ es:
        singletonbag(e) ≡ nonemptybag(e, emptybag),
        unionbag(emptybag, b) ≡ b,
        unionbag(nonemptybag(e, b), b') ≡ nonemptybag(e, unionbag (b, b')),
        without(emptybag, e) ≡ emptybag,
        without(nonemptybag(e, b), e') ≡
            if e = e' then b else nonemptybag(e, without(b, e')) fi,
        elembag(e, emptybag) ≡ false,
        elembag(e, nonemptybag(e', b)) ≡ e = e' ∨ elembag(e, b),
        equals(nonemptybag(e, b), b') ≡
```

elembag(e, b') $\wedge$ equals(b, without(b', e)),
minbag(elementsof(e:es)) $\equiv$ minimum(e:es),
elementsof([ ]) $\equiv$ emptybag,
elementsof(e:es) $\equiv$ nonemptybag(e, elementsof(es)),
equals(b, b') $\equiv$ equals(unionbag(b", b), unionbag(b", b')),
equals(b, b') $\wedge$ e = minbag(b) $\equiv$ equals(b, b') $\wedge$ e = minbag(b')

endoftype

Several parts of the specification are already implemented in Haskell. More precisely, we assume (monomorphic) representations for BOOL, MSEQU, and elem with $\leq$ and $=$. In the remaining part of this section we prove a few facts about the specification of bags. The important result of this section is Theorem A.10 that explains the adequacy of the specification.

Lemma A.1. *All bag-valued terms can be generated by emptybag and nonemptybag, i.e., for every bag b there exists a sequence $e_1, e_2, \ldots, e_k$ and a sequence $b_0, b_1, b_2, \ldots, b_k$ with*

$$\begin{array}{lcll} b_0 & = & \textit{emptybag} & \\ b_i & = & \textit{nonemptybag}(e_i,\ b_{i-1}) & \quad (\textit{for } 1 \leq i \leq k) \\ b_k & \equiv & b & \end{array}$$

We call b_k the generating term of b.

Proof. We proceed by structural induction over bag. Let b be an arbitrary bag-valued term.

- If b=emptybag, emptybag is the generating term of b.
- If b=nonemptybag(e, b') for some e and b', due to the induction hypothesis we can generate b' by emptybag and nonemptybag. Let c' be the generating term of b'. Then, nonemptybag(e, c') is the generating term of b.
- If b=singletonbag(e) for some e, b$\equiv$nonemptybag(e, emptybag) by the specification of bags, hence the latter can be used as the generating term of b.
- If b=unionbag(b', b") for some b' and b", due to the induction hypothesis we can generate b' and b" by emptybag and nonemptybag. Let c' be the generating term of b'. We prove by induction over the structure of c' that unionbag(c', b") can be generated by emptybag and nonemptybag. If c'=emptybag, b$\equiv$unionbag(emptybag, b")$\equiv$b" by the specification of bags. If c'=nonemptybag(e, c) for some e and c, by the specification of bags b$\equiv$unionbag(nonemptybag(e, c), b")$\equiv$nonemptybag(e, unionbag(c, b")). By the induction hypothesis, unionbag(c, b") can be generated by emptybag and nonemptybag, hence also b.
- If b=without(b', e) for some b' and e, due to the induction hypothesis we can generate b' by emptybag and nonemptybag. Let c' be the generating term of b'. A similar proof by induction over the structure of c' using the laws of the specification concerning without can be performed.
- If b=elementsof(es) for some es, a proof by induction over the structure of es using the defining equations of elementsof can be performed.

No other possibility exists to construct b. □

Lemma A.2. *In the specification of Bag, elementsof is surjective.*

Proof. Let b be an arbitrary element of the sort bag. By Lemma A.1, b can be generated by emptybag and nonemptybag, i.e., there exists a sequence $e_1, e_2, \ldots, e_k$ and a sequence $b_0, b_1, b_2, \ldots, b_k$ with

$$\begin{array}{lcll} b_0 & = & \text{emptybag} & \\ b_i & = & \text{nonemptybag}(e_i,\, b_{i-1}) & \quad (\text{for } 1 \le i \le k) \\ b_k & \equiv & \text{b} & \end{array}$$

Define

$$\begin{array}{lcll} es_0 & =_{\text{def}} & [\,] & \\ es_i & =_{\text{def}} & e_i : es_{i-1} & \quad (\text{for } 1 \le i \le k) \end{array}$$

We prove by induction that $\forall i \in \{0, \ldots, k\} : b_i \equiv \text{elementsof}(es_i)$. For $i = 0$ we have $b_0 = \text{emptybag} \equiv \text{elementsof}([]) = \text{elementsof}(es_0)$. Assume now that $i \ge 1$ and $\forall j \in \{0, \ldots, i-1\} : b_j \equiv \text{elementsof}(es_j)$. We calculate

$$\begin{array}{ll} & b_i \\ = & \text{nonemptybag}(e_i,\, b_{i-1}) \\ \equiv & \text{nonemptybag}(e_i,\, \text{elementsof}(es_{i-1})) \\ \equiv & \text{elementsof}(e_i : es_{i-1}) \\ = & \text{elementsof}(es_i). \end{array}$$

This finishes the induction and, therefore, b $\equiv b_k \equiv \text{elementsof}(es_k)$. □

Lemma A.3. *Let xs and ys be arbitrary sequences. Then,*

$$\textit{xs is a permutation of ys} \iff \textit{xs} \sim \textit{ys} \equiv \textit{true}$$

where xs ∼ ys is short for equals(elementsof(xs), elementsof(ys)).

Proof. For the forward direction we can reduce a permutation to a series of inversions of two adjacent elements and then apply transitivity of equals. Therefore, assume that ys can be obtained from xs by swapping two adjacent elements. By the laws of the specification of bags we may write both elementsof(xs) and elementsof(ys) as a sequence of unionbags of singletonbags. By associativity and commutativity of unionbag, we can perform the swap operation, hence xs ≡ ys. The first part of the proof is finished by reflexivity of equals.

For the backward direction we proceed by induction over the length of xs and ys and distinguish four cases.

- If xs=[ ] and ys=[ ], xs ∼ ys ≡ true by reflexivity of equals and xs is a permutation of ys.
- If xs≠[ ] and ys=[ ], xs ∼ ys ≡ false by the laws for equals and elembag.
- If xs=[ ] and ys≠[ ], xs ∼ ys ≡ false by commutativity of equals and the preceding case.
- Otherwise, let xs=x:xs' and ys=y:ys' with xs ∼ ys ≡ true. Then, by the defining equation of equals,

$$\text{elembag(x, elementsof(ys))} \wedge \text{equals(xs', without(elementsof(ys), x))} \equiv \text{true}.$$

By induction over the length of ys using the laws for elembag and without we can prove that

elembag(x, elementsof(ys)) $\equiv$ x $\in$ ys, and
without(elementsof(ys), x) $\equiv$ elementsof(ys $\setminus$ x)

where $\setminus$ and $\in$ denote the element removal and testing functions over lists. Hence, by the induction hypothesis, x $\in$ ys and xs' is a permutation of ys $\setminus$ x. Therefore, xs is a permutation of ys.

□

Lemma A.4. *The following part of the specification of bags is a consequence of the remaining laws:*

1. **include** ASSOC(bag, unionbag), i.e.,
   unionbag(unionbag(b, b'), b") $\equiv$ unionbag(b, unionbag(b', b")).
2. equals(b, b') $\equiv$ equals(unionbag(b", b), unionbag(b", b')).
3. equals(b, b') $\wedge$ e = minbag(b) $\equiv$ equals(b, b') $\wedge$ e = minbag(b').

Proof. We discuss the three laws in turn.

1. Let c be the generating term of b that exists according to Lemma A.1. We prove the associativity by induction over the structure of c. If c=emptybag,

$$\begin{array}{ll} & \text{unionbag(unionbag(b, b'), b")} \\ \equiv & \text{unionbag(unionbag(emptybag, b'), b")} \\ \equiv & \text{unionbag(b', b")} \\ \equiv & \text{unionbag(emptybag, unionbag(b', b"))} \\ \equiv & \text{unionbag(b, unionbag(b', b")).} \end{array}$$

   If c=nonemptybag(e, c') for some e and c',

$$\begin{array}{ll} & \text{unionbag(unionbag(b, b'), b")} \\ \equiv & \text{unionbag(unionbag(nonemptybag(e, c'), b'), b")} \\ \equiv & \text{unionbag(nonemptybag(e, unionbag(c', b')), b")} \\ \equiv & \text{nonemptybag(e, unionbag(unionbag(c', b'), b"))} \\ \equiv & \text{nonemptybag(e, unionbag(c', unionbag(b', b")))} \\ \equiv & \text{unionbag(nonemptybag(e, c'), unionbag(b', b"))} \\ \equiv & \text{unionbag(b, unionbag(b', b")).} \end{array}$$

2. Let c" be the generating term of b" that exists according to Lemma A.1. We prove the proposition by induction over the structure of c". If c"=emptybag,

$$\begin{array}{ll} & \text{equals(unionbag(b", b), unionbag(b", b'))} \\ \equiv & \text{equals(unionbag(emptybag, b), unionbag(emptybag, b'))} \\ \equiv & \text{equals(b, b').} \end{array}$$

   If c"=nonemptybag(e, c) for some e and c,

$$\begin{array}{ll}
 & \text{equals(unionbag(b'', b), unionbag(b'', b'))} \\
\equiv & \text{equals(unionbag(nonemptybag(e, c), b),} \\
 & \quad \text{unionbag(nonemptybag(e, c), b'))} \\
\equiv & \text{equals(nonemptybag(e, unionbag(c, b)),} \\
 & \quad \text{nonemptybag(e, unionbag(c, b')))} \\
\equiv & \text{elembag(e, nonemptybag(e, unionbag(c, b'))) } \wedge \\
 & \quad \text{equals(unionbag(c, b),} \\
 & \qquad \text{without(nonemptybag(e, unionbag(c, b')), e))} \\
\equiv & (\text{e} = \text{e} \vee \ldots) \wedge \\
 & \quad \text{equals(unionbag(c, b), } \textbf{if } \text{e} = \text{e } \textbf{then } \text{unionbag(c, b') } \textbf{else } \ldots \textbf{fi}) \\
\equiv & \text{true} \wedge \text{equals(unionbag(c, b), unionbag(c, b'))} \\
\equiv & \text{equals(b, b')}.
\end{array}$$

3. If equals(b, b') ≡ false, both sides are ≡ false. Otherwise, assume that equals(b, b') ≡ true. By Lemma A.2, there exist sequences xs and ys with elementsof(xs) ≡ b and elementsof(ys) ≡ b'. By Lemma A.3, xs is a permutation of ys. Therefore, according to the specification of bags since b and b' are not empty,

$$\begin{array}{ll}
 & \text{minbag(b)} \\
\equiv & \text{minbag(elementsof(xs))} \\
\equiv & \text{minimum(xs)} \\
\equiv & \text{minimum(ys)} \\
\equiv & \text{minbag(elementsof(ys))} \\
\equiv & \text{minbag(b')}.
\end{array}$$

Note that the proof of Lemma A.3 uses the associativity of unionbag, a fact that can be proved from the laws concerning unionbag (see part 1 of this proof).

□

Lemma A.5. *In the specification of Bag the law*

minbag(elementsof(e:es)) ≡ minimum(e:es)

is equivalent to the two laws

1. *minbag(nonemptybag(e, emptybag)) ≡ e,*
2. *minbag(nonemptybag(e, nonemptybag(e', b))) ≡ min(e, minbag(nonemptybag(e', b)))*

where min denotes the minimum function for two elements.

Proof. Assume that the laws 1 and 2 hold. We proceed by structural induction on (e:es). If es=[ ] then

$$\begin{array}{ll}
 & \text{minbag(elementsof(e:[ ]))} \\
\equiv & \text{minbag(nonemptybag(e, elementsof([ ])))} \\
\equiv & \text{minbag(nonemptybag(e, emptybag))} \\
\equiv & \text{e} \\
\equiv & \text{minimum(e:[ ])}.
\end{array}$$

If es=(e':es') then

minbag(elementsof(e:e':es'))
$\equiv$ minbag(nonemptybag(e, elementsof(e':es')))
$\equiv$ minbag(nonemptybag(e, nonemptybag(e', elementsof(es'))))
$\equiv$ min(e, minbag(nonemptybag(e', elementsof(es'))))
$\equiv$ min(e, minbag(elementsof(e':es')))
$\equiv$ min(e, minimum(e':es'))
$\equiv$ minimum(e:e':es').

Assume that the law of the specification holds. First,

minbag(nonemptybag(e, emptybag))
$\equiv$ minbag(elementsof(e:[ ]))
$\equiv$ minimum(e:[ ])
$\equiv$ e.

Second, for all bags b by Lemma A.2 there exists a list es' such that elementsof(es') = b. Then,

minbag(nonemptybag(e, nonemptybag(e', b)))
$\equiv$ minbag(nonemptybag(e, nonemptybag(e', elementsof(es'))))
$\equiv$ minbag(nonemptybag(e, elementsof(e':es')))
$\equiv$ minbag(elementsof(e:e':es'))
$\equiv$ minimum(e:e':es')
$\equiv$ min(e, minimum(e':es'))
$\equiv$ min(e, minbag(elementsof(e':es')))
$\equiv$ min(e, minbag(nonemptybag(e', elementsof(es'))))
$\equiv$ min(e, minbag(nonemptybag(e', b))).

$\square$

Lemma A.6. *In the specification of Bag, for all elements e, e' and bags b, b'*

minbag(unionbag(nonemptybag(e, b), nonemptybag(e', b'))) $\equiv$
min(minbag(nonemptybag(e, b)), minbag(nonemptybag(e', b'))).

Proof. We prove that for all elements e, e' and bags b' and sequences of elements es

minbag(unionbag(nonemptybag(e, elementsof(es)), nonemptybag(e', b'))) $\equiv$
min(minbag(nonemptybag(e, elementsof(es))), minbag(nonemptybag(e', b'))),

from which the claim follows by Lemma A.2. The proof is by structural induction on es using Lemma A.5. If es=[ ], we calculate

minbag(unionbag(nonemptybag(e, elementsof(es)), nonemptybag(e', b')))
= minbag(unionbag(nonemptybag(e, elementsof([ ])), nonemptybag(e', b')))
$\equiv$ minbag(unionbag(nonemptybag(e, emptybag), nonemptybag(e', b')))
$\equiv$ minbag(nonemptybag(e, unionbag(emptybag, nonemptybag(e', b'))))
$\equiv$ minbag(nonemptybag(e, nonemptybag(e', b')))
$\equiv$ min(e, minbag(nonemptybag(e', b')))
$\equiv$ min(minbag(nonemptybag(e, emptybag)), minbag(nonemptybag(e', b')))
$\equiv$ min(minbag(nonemptybag(e, elementsof([ ]))),
    minbag(nonemptybag(e', b')))
= min(minbag(nonemptybag(e, elementsof(es))),
    minbag(nonemptybag(e', b'))).

Otherwise, let es=e”:es” and calculate

$$\begin{array}{ll}
 & \text{minbag(unionbag(nonemptybag(e, elementsof(es)), nonemptybag(e', b')))} \\
= & \text{minbag(unionbag(nonemptybag(e, elementsof(e”:es”)),} \\
 & \quad \text{nonemptybag(e', b')))} \\
\equiv & \text{minbag(nonemptybag(e, unionbag(elementsof(e”:es”),} \\
 & \quad \text{nonemptybag(e', b'))))} \\
\equiv & \text{minbag(nonemptybag(e, unionbag(nonemptybag(e”, elementsof(es”)),} \\
 & \quad \text{nonemptybag(e', b'))))} \\
\equiv & \text{minbag(nonemptybag(e, nonemptybag(e”, unionbag(elementsof(es”),} \\
 & \quad \text{nonemptybag(e', b')))))} \\
\equiv & \text{min(e, minbag(nonemptybag(e”, unionbag(elementsof(es”),} \\
 & \quad \text{nonemptybag(e', b')))))} \\
\equiv & \text{min(e, minbag(unionbag(nonemptybag(e”, elementsof(es”)),} \\
 & \quad \text{nonemptybag(e', b'))))} \\
\equiv & \text{min(e, min(minbag(nonemptybag(e”, elementsof(es”))),} \\
 & \quad \text{minbag(nonemptybag(e', b'))))} \\
\equiv & \text{min(min(e, minbag(nonemptybag(e”, elementsof(es”)))),} \\
 & \quad \text{minbag(nonemptybag(e', b')))} \\
\equiv & \text{min(minbag(nonemptybag(e, nonemptybag(e”, elementsof(es”)))),} \\
 & \quad \text{minbag(nonemptybag(e', b')))} \\
\equiv & \text{min(minbag(nonemptybag(e, elementsof(e”:es”))),} \\
 & \quad \text{minbag(nonemptybag(e', b')))} \\
= & \text{min(minbag(nonemptybag(e, elementsof(es))),} \\
 & \quad \text{minbag(nonemptybag(e', b'))).}
\end{array}$$

□

Lemma A.7. *In the specification of Bag the law*

equals(nonemptybag(e, b), b') $\equiv$ elembag(e, b') $\wedge$ equals(b, without(b', e))

is equivalent to the four laws

1. *equals(emptybag, emptybag) $\equiv$ true,*
2. *equals(emptybag, nonemptybag(e, b)) $\equiv$ false,*
3. *equals(nonemptybag(e, b), emptybag) $\equiv$ false,*
4. *equals(nonemptybag(e, b), b') $\equiv$ elembag(e, b') $\wedge$ equals(b, without(b', e)),*

the last of which is retained.

Proof. Since law 4 is retained, it suffices to show that laws 1–3 are implied by the specification of Bag. Law 1 follows from reflexivity included by EQUIV(bag, equals). Law 2 follows from law 3 and commutativity included by EQUIV(bag, equals). Law 3 follows from the specification by

$$\begin{array}{ll} & \text{equals(nonemptybag(e, b), emptybag)} \\ \equiv & \text{elembag(e, emptybag)} \wedge \text{equals(b, without(emptybag, e))} \\ \equiv & \text{false} \wedge \text{equals(b, without(emptybag, e))} \\ \equiv & \text{false.} \end{array}$$

□

Theorem A.8. *All models of Bag are behaviourally equivalent.*

Proof. By replacing the indicated laws according to Lemmas A.5 and A.7, we get an equivalent specification that is sufficiently complete. By [Wir90, Fact 5.4.4] it follows that all models of Bag are behaviourally equivalent. □

Lemma A.9. *The "classical" mathematical algebra B of bags defined on top of the "classical" algebras of booleans and lists by*

$$\begin{array}{lll} \text{bag}^B & =_{\text{def}} & \text{elem} \to \mathbb{N} \\ \text{emptybag}^B & =_{\text{def}} & \lambda x.0 \\ \text{nonemptybag}^B(e,b) & =_{\text{def}} & \lambda x.\textbf{if } x = e \textbf{ then } b(x)+1 \textbf{ else } b(x) \textbf{ fi} \\ \text{singletonbag}^B(e) & =_{\text{def}} & \lambda x.\textbf{if } x = e \textbf{ then } 1 \textbf{ else } 0 \textbf{ fi} \\ \text{unionbag}^B(b,b') & =_{\text{def}} & \lambda x.b(x) + b'(x) \\ \text{equals}^B(b,b') & =_{\text{def}} & \forall x \in \text{elem} : b(x) = b'(x) \\ \text{elembag}^B(e,b) & =_{\text{def}} & b(e) \geq 1 \\ \text{without}^B(b,e) & =_{\text{def}} & \lambda x.\textbf{if } x = e \wedge b(x) \geq 1 \textbf{ then } b(x) - 1 \textbf{ else } b(x) \textbf{ fi} \\ \text{minbag}^B(b) & =_{\text{def}} & \min\{x \in \text{elem} : b(x) \geq 1\} \\ \text{elementsof}^B([\,]) & =_{\text{def}} & \text{emptybag}^B \\ \text{elementsof}^B(e : es) & =_{\text{def}} & \text{nonemptybag}^B(e, \text{elementsof}^B(es)) \end{array}$$

is a model of the specification Bag.

Proof. We have to show the validity of the laws of the specification. Three of them have already been proved to be consequences of the other ones in Lemma A.4. We start with the included laws.

$$\begin{array}{ll} & \text{unionbag}^B(b,b') \\ = & \lambda x.b(x) + b'(x) \\ = & \lambda x.b'(x) + b(x) \\ = & \text{unionbag}^B(b',b). \end{array}$$

$$\begin{array}{ll} & \text{equals}^B(b,b) \\ = & \forall x \in \text{elem} : b(x) = b(x) \\ = & \forall x \in \text{elem} : \text{true} \\ = & \text{true.} \end{array}$$

$$\begin{array}{ll} & \text{equals}^B(b,b') \\ = & \forall x \in \text{elem} : b(x) = b'(x) \\ = & \forall x \in \text{elem} : b'(x) = b(x) \\ = & \text{equals}^B(b',b). \end{array}$$

$$\begin{array}{ll}
& \text{equals}^B(b,b') \wedge \text{equals}^B(b',b'') \;\Rightarrow\; \text{equals}^B(b,b'') \\
= & (\forall x \in \text{elem} : b(x) = b'(x)) \wedge (\forall x \in \text{elem} : b'(x) = b''(x)) \;\Rightarrow\; \text{equals}^B(b,b'') \\
= & (\forall x \in \text{elem} : b(x) = b'(x) \wedge b'(x) = b''(x)) \;\Rightarrow\; \text{equals}^B(b,b'') \\
= & (\forall x \in \text{elem} : b(x) = b'(x) \wedge b(x) = b''(x)) \;\Rightarrow\; \text{equals}^B(b,b'') \\
= & (\forall x \in \text{elem} : b(x) = b'(x)) \wedge (\forall x \in \text{elem} : b(x) = b''(x)) \;\Rightarrow\; \text{equals}^B(b,b'') \\
= & \text{equals}^B(b,b') \wedge \text{equals}^B(b,b'') \;\Rightarrow\; \text{equals}^B(b,b'') \\
= & \text{true.}
\end{array}$$

We continue with the laws that were not included.

$$\begin{array}{ll}
& \text{singletonbag}^B(e) \\
= & \lambda x.\textbf{if } x = e \textbf{ then } 1 \textbf{ else } 0 \textbf{ fi} \\
= & \lambda x.\textbf{if } x = e \textbf{ then } 0 + 1 \textbf{ else } 0 \textbf{ fi} \\
= & \lambda x.\textbf{if } x = e \textbf{ then } (\lambda x.0)(x) + 1 \textbf{ else } (\lambda x.0)(x) \textbf{ fi} \\
= & \lambda x.\textbf{if } x = e \textbf{ then } \text{emptybag}^B(x) + 1 \textbf{ else } \text{emptybag}^B(x) \textbf{ fi} \\
= & \text{nonemptybag}^B(e, \text{emptybag}^B).
\end{array}$$

$$\begin{array}{ll}
& \text{unionbag}^B(\text{emptybag}^B, b) \\
= & \lambda x.\text{emptybag}^B(x) + b(x) \\
= & \lambda x.(\lambda x.0)(x) + b(x) \\
= & \lambda x.0 + b(x) \\
= & \lambda x.b(x) \\
= & b.
\end{array}$$

$$\begin{array}{ll}
& \text{unionbag}^B(\text{nonemptybag}^B(e,b), b') \\
= & \lambda x.\text{nonemptybag}^B(e,b)(x) + b'(x) \\
= & \lambda x.(\lambda x.\textbf{if } x = e \textbf{ then } b(x) + 1 \textbf{ else } b(x) \textbf{ fi})(x) + b'(x) \\
= & \lambda x.\textbf{if } x = e \textbf{ then } b(x) + 1 \textbf{ else } b(x) \textbf{ fi} + b'(x) \\
= & \lambda x.\textbf{if } x = e \textbf{ then } b(x) + 1 + b'(x) \textbf{ else } b(x) + b'(x) \textbf{ fi} \\
= & \lambda x.\textbf{if } x = e \textbf{ then } (\lambda x.b(x) + b'(x))(x) + 1 \textbf{ else } (\lambda x.b(x) + b'(x))(x) \textbf{ fi} \\
= & \lambda x.\textbf{if } x = e \textbf{ then } \text{unionbag}^B(b,b')(x) + 1 \textbf{ else } \text{unionbag}^B(b,b')(x) \textbf{ fi} \\
= & \text{nonemptybag}^B(e, \text{unionbag}^B(b,b')).
\end{array}$$

$$\begin{array}{ll}
& \text{equals}^B(\text{nonemptybag}^B(e,b), b') \\
= & \text{equals}^B(\lambda x.\textbf{if } x = e \textbf{ then } b(x) + 1 \textbf{ else } b(x) \textbf{ fi}, b') \\
= & \forall x \in \text{elem} : \textbf{if } x = e \textbf{ then } b(x) + 1 \textbf{ else } b(x) \textbf{ fi} = b'(x) \\
= & b'(e) \geq 1 \wedge \forall x \in \text{elem} : \textbf{if } x = e \textbf{ then } b(x) + 1 \textbf{ else } b(x) \textbf{ fi} = b'(x) \\
= & b'(e) \geq 1 \wedge \forall x \in \text{elem} : \textbf{if } x = e \textbf{ then } b(x) + 1 = b'(x) \textbf{ else } b(x) = b'(x) \textbf{ fi} \\
= & b'(e) \geq 1 \wedge \forall x \in \text{elem} : \textbf{if } x = e \wedge b'(x) \geq 1 \textbf{ then } b(x) + 1 = b'(x) \\
& \qquad \textbf{else } b(x) = b'(x) \textbf{ fi} \\
= & b'(e) \geq 1 \wedge \forall x \in \text{elem} : \textbf{if } x = e \wedge b'(x) \geq 1 \textbf{ then } b(x) = b'(x) - 1 \\
& \qquad \textbf{else } b(x) = b'(x) \textbf{ fi} \\
= & b'(e) \geq 1 \wedge \forall x \in \text{elem} : b(x) = \textbf{if } x = e \wedge b'(x) \geq 1 \textbf{ then } b'(x) - 1 \\
& \qquad \textbf{else } b'(x) \textbf{ fi} \\
= & b'(e) \geq 1 \wedge \text{equals}^B(b, \lambda x.\textbf{if } x = e \wedge b'(x) \geq 1 \textbf{ then } b'(x) - 1 \textbf{ else } b'(x) \textbf{ fi}) \\
= & b'(e) \geq 1 \wedge \text{equals}^B(b, \text{without}^B(b', e)) \\
= & \text{elembag}^B(e,b') \wedge \text{equals}^B(b, \text{without}^B(b', e)).
\end{array}$$

$$
\begin{array}{ll}
 & \mathrm{elembag}^B(e, \mathrm{emptybag}^B) \\
= & \mathrm{emptybag}^B(e) \geq 1 \\
= & 0 \geq 1 \\
= & \mathrm{false}.
\end{array}
$$

$$
\begin{array}{ll}
 & \mathrm{elembag}^B(e, \mathrm{nonemptybag}^B(e', b)) \\
= & \mathrm{nonemptybag}^B(e', b)(e) \geq 1 \\
= & (\lambda x.\mathbf{if}\ x = e'\ \mathbf{then}\ b(x) + 1\ \mathbf{else}\ b(x)\ \mathbf{fi})(e) \geq 1 \\
= & \mathbf{if}\ e = e'\ \mathbf{then}\ b(e) + 1\ \mathbf{else}\ b(e)\ \mathbf{fi} \geq 1 \\
= & \mathbf{if}\ e = e'\ \mathbf{then}\ b(e) + 1 \geq 1\ \mathbf{else}\ b(e) \geq 1\ \mathbf{fi} \\
= & \mathbf{if}\ e = e'\ \mathbf{then}\ \mathrm{true}\ \mathbf{else}\ b(e) \geq 1\ \mathbf{fi} \\
= & e = e' \vee b(e) \geq 1 \\
= & e = e' \vee \mathrm{elembag}^B(e, b).
\end{array}
$$

$$
\begin{array}{ll}
 & \mathrm{without}^B(\mathrm{emptybag}^B, e) \\
= & \mathrm{without}^B(\lambda x.0, e) \\
= & \lambda x.\mathbf{if}\ x = e \wedge (\lambda x.0)(x) \geq 1\ \mathbf{then}\ (\lambda x.0)(x) - 1\ \mathbf{else}\ (\lambda x.0)(x)\ \mathbf{fi} \\
= & \lambda x.\mathbf{if}\ x = e \wedge 0 \geq 1\ \mathbf{then}\ 0 - 1\ \mathbf{else}\ 0\ \mathbf{fi} \\
= & \lambda x.\mathbf{if}\ \mathrm{false}\ \mathbf{then}\ 0 - 1\ \mathbf{else}\ 0\ \mathbf{fi} \\
= & \lambda x.0 \\
= & \mathrm{emptybag}^B.
\end{array}
$$

$$
\begin{array}{ll}
 & \mathrm{without}^B(\mathrm{nonemptybag}^B(e, b), e') \\
= & \mathbf{if}\ e = e'\ \mathbf{then}\ \mathrm{without}^B(\mathrm{nonemptybag}^B(e, b), e) \\
 & \mathbf{else}\ \mathrm{without}^B(\mathrm{nonemptybag}^B(e, b), e')\ \mathbf{fi} \\
= & \mathbf{if}\ e = e'\ \mathbf{then}\ \lambda x.\mathbf{if}\ x = e \wedge \mathrm{nonemptybag}^B(e, b)(x) \geq 1 \\
 & \quad \mathbf{then}\ \mathrm{nonemptybag}^B(e, b)(x) - 1\ \mathbf{else}\ \mathrm{nonemptybag}^B(e, b)(x)\ \mathbf{fi} \\
 & \mathbf{else}\ \mathrm{without}^B(\mathrm{nonemptybag}^B(e, b), e')\ \mathbf{fi} \\
= & \mathbf{if}\ e = e'\ \mathbf{then}\ \lambda x.\mathbf{if}\ x = e \wedge (\mathbf{if}\ x = e\ \mathbf{then}\ b(x) + 1\ \mathbf{else}\ b(x)\ \mathbf{fi}) \geq 1 \\
 & \quad \mathbf{then}\ \mathrm{nonemptybag}^B(e, b)(x) - 1\ \mathbf{else}\ \mathrm{nonemptybag}^B(e, b)(x)\ \mathbf{fi} \\
 & \mathbf{else}\ \mathrm{without}^B(\mathrm{nonemptybag}^B(e, b), e')\ \mathbf{fi} \\
= & \mathbf{if}\ e = e'\ \mathbf{then}\ \lambda x.\mathbf{if}\ x = e \wedge b(x) + 1 \geq 1\ \mathbf{then}\ \mathrm{nonemptybag}^B(e, b)(x) - 1 \\
 & \mathbf{else}\ \mathrm{nonemptybag}^B(e, b)(x)\ \mathbf{fi}\ \mathbf{else}\ \mathrm{without}^B(\mathrm{nonemptybag}^B(e, b), e')\ \mathbf{fi} \\
= & \mathbf{if}\ e = e'\ \mathbf{then}\ \lambda x.\mathbf{if}\ x = e\ \mathbf{then}\ \mathrm{nonemptybag}^B(e, b)(e) - 1 \\
 & \mathbf{else}\ \mathrm{nonemptybag}^B(e, b)(x)\ \mathbf{fi}\ \mathbf{else}\ \mathrm{without}^B(\mathrm{nonemptybag}^B(e, b), e')\ \mathbf{fi} \\
= & \mathbf{if}\ e = e'\ \mathbf{then}\ \lambda x.\mathbf{if}\ x = e\ \mathbf{then}\ \mathbf{if}\ e = e\ \mathbf{then}\ b(e) + 1\ \mathbf{else}\ b(e)\ \mathbf{fi} - 1 \\
 & \mathbf{else}\ \mathrm{nonemptybag}^B(e, b)(x)\ \mathbf{fi}\ \mathbf{else}\ \mathrm{without}^B(\mathrm{nonemptybag}^B(e, b), e')\ \mathbf{fi} \\
= & \mathbf{if}\ e = e'\ \mathbf{then}\ \lambda x.\mathbf{if}\ x = e\ \mathbf{then}\ b(e) + 1 - 1 \\
 & \mathbf{else}\ \mathrm{nonemptybag}^B(e, b)(x)\ \mathbf{fi}\ \mathbf{else}\ \mathrm{without}^B(\mathrm{nonemptybag}^B(e, b), e')\ \mathbf{fi} \\
= & \mathbf{if}\ e = e'\ \mathbf{then}\ \lambda x.\mathbf{if}\ x = e\ \mathbf{then}\ b(e)\ \mathbf{else}\ \mathbf{if}\ x = e\ \mathbf{then}\ b(x) + 1\ \mathbf{else}\ b(x)\ \mathbf{fi}\ \mathbf{fi} \\
 & \mathbf{else}\ \mathrm{without}^B(\mathrm{nonemptybag}^B(e, b), e')\ \mathbf{fi} \\
= & \mathbf{if}\ e = e'\ \mathbf{then}\ \lambda x.\mathbf{if}\ x = e\ \mathbf{then}\ b(x)\ \mathbf{else}\ b(x)\ \mathbf{fi} \\
 & \mathbf{else}\ \mathrm{without}^B(\mathrm{nonemptybag}^B(e, b), e')\ \mathbf{fi} \\
= & \mathbf{if}\ e = e'\ \mathbf{then}\ \lambda x.b(x)\ \mathbf{else}\ \lambda x.\mathrm{without}^B(\mathrm{nonemptybag}^B(e, b), e')(x)\ \mathbf{fi} \\
= & \mathbf{if}\ e = e'\ \mathbf{then}\ b\ \mathbf{else}\ \lambda x.\mathbf{if}\ x = e\ \mathbf{then}\ \mathrm{without}^B(\mathrm{nonemptybag}^B(e, b), e')(e) \\
 & \mathbf{else}\ \mathrm{without}^B(\mathrm{nonemptybag}^B(e, b), e')(x)\ \mathbf{fi}\ \mathbf{fi} \\
= & \mathbf{if}\ e = e'\ \mathbf{then}\ b\ \mathbf{else}\ \lambda x.\mathbf{if}\ x = e\ \mathbf{then}\ \mathbf{if}\ e = e' \wedge \mathrm{nonemptybag}^B(e, b)(e) \geq 1 \\
 & \quad \mathbf{then}\ \mathrm{nonemptybag}^B(e, b)(e) - 1\ \mathbf{else}\ \mathrm{nonemptybag}^B(e, b)(e)\ \mathbf{fi}
\end{array}
$$

$$
\begin{array}{ll}
 & \textbf{else } \mathrm{without}^B(\mathrm{nonemptybag}^B(e,b),e')(x) \textbf{ fi fi} \\
= & \textbf{if } e = e' \textbf{ then } b \textbf{ else } \lambda x.\textbf{if } x = e \\
 & \textbf{then if } \mathrm{false} \textbf{ then } \mathrm{nonemptybag}^B(e,b)(e) - 1 \textbf{ else } \mathrm{nonemptybag}^B(e,b)(e) \textbf{ fi} \\
 & \textbf{else } \mathrm{without}^B(\mathrm{nonemptybag}^B(e,b),e')(x) \textbf{ fi fi} \\
= & \textbf{if } e = e' \textbf{ then } b \textbf{ else } \lambda x.\textbf{if } x = e \textbf{ then } \mathrm{nonemptybag}^B(e,b)(e) \\
 & \textbf{else } \mathrm{without}^B(\mathrm{nonemptybag}^B(e,b),e')(x) \textbf{ fi fi} \\
= & \textbf{if } e = e' \textbf{ then } b \textbf{ else } \lambda x.\textbf{if } x = e \textbf{ then if } e = e \textbf{ then } b(e) + 1 \textbf{ else } b(e) \textbf{ fi} \\
 & \textbf{else } \mathrm{without}^B(\mathrm{nonemptybag}^B(e,b),e')(x) \textbf{ fi fi} \\
= & \textbf{if } e = e' \textbf{ then } b \textbf{ else } \lambda x.\textbf{if } x = e \textbf{ then } b(x) + 1 \\
 & \textbf{else } \mathrm{without}^B(\mathrm{nonemptybag}^B(e,b),e')(x) \textbf{ fi fi} \\
= & \textbf{if } e = e' \textbf{ then } b \textbf{ else } \lambda x.\textbf{if } x = e \textbf{ then } b(x) + 1 \\
 & \textbf{else if } x = e' \wedge \mathrm{nonemptybag}^B(e,b)(e') \geq 1 \\
 & \textbf{then } \mathrm{nonemptybag}^B(e,b)(x) - 1 \textbf{ else } \mathrm{nonemptybag}^B(e,b)(x) \textbf{ fi fi fi} \\
= & \textbf{if } e = e' \textbf{ then } b \textbf{ else } \lambda x.\textbf{if } x = e \textbf{ then } b(x) + 1 \\
 & \textbf{else if } x = e' \wedge (\textbf{if } e' = e \textbf{ then } b(e') + 1 \textbf{ else } b(e') \textbf{ fi}) \geq 1 \\
 & \textbf{then } \mathrm{nonemptybag}^B(e,b)(x) - 1 \textbf{ else } \mathrm{nonemptybag}^B(e,b)(x) \textbf{ fi fi fi} \\
= & \textbf{if } e = e' \textbf{ then } b \textbf{ else } \lambda x.\textbf{if } x = e \textbf{ then } b(x) + 1 \textbf{ else if } x = e' \wedge b(e') \geq 1 \\
 & \textbf{then } \mathrm{nonemptybag}^B(e,b)(e') - 1 \textbf{ else } \mathrm{nonemptybag}^B(e,b)(x) \textbf{ fi fi fi} \\
= & \textbf{if } e = e' \textbf{ then } b \textbf{ else } \lambda x.\textbf{if } x = e \textbf{ then } b(x) + 1 \textbf{ else if } x = e' \wedge b(e') \geq 1 \\
 & \textbf{then if } e' = e \textbf{ then } b(e') + 1 \textbf{ else } b(e') \textbf{ fi} - 1 \\
 & \textbf{else } \mathrm{nonemptybag}^B(e,b)(x) \textbf{ fi fi fi} \\
= & \textbf{if } e = e' \textbf{ then } b \textbf{ else } \lambda x.\textbf{if } x = e \textbf{ then } b(x) + 1 \textbf{ else if } x = e' \wedge b(e') \geq 1 \\
 & \textbf{then } b(e') - 1 \textbf{ else if } x = e \textbf{ then } b(x) + 1 \textbf{ else } b(x) \textbf{ fi fi fi fi} \\
= & \textbf{if } e = e' \textbf{ then } b \textbf{ else } \lambda x.\textbf{if } x = e \textbf{ then } b(x) + 1 \textbf{ else if } x = e' \wedge b(x) \geq 1 \\
 & \textbf{then } b(x) - 1 \textbf{ else } b(x) \textbf{ fi fi fi} \\
= & \textbf{if } e = e' \textbf{ then } b \textbf{ else } \lambda x.\textbf{if } x = e \textbf{ then } b(x) + 1 \textbf{ else } \mathrm{without}^B(b,e')(x) \textbf{ fi fi} \\
= & \textbf{if } e = e' \textbf{ then } b \textbf{ else } \lambda x.\textbf{if } x = e \textbf{ then if } \mathrm{false} \wedge b(x) \geq 1 \\
 & \textbf{then } b(x) - 1 \textbf{ else } b(x) \textbf{ fi} + 1 \textbf{ else } \mathrm{without}^B(b,e')(x) \textbf{ fi fi} \\
= & \textbf{if } e = e' \textbf{ then } b \textbf{ else } \lambda x.\textbf{if } x = e \textbf{ then if } x = e' \wedge b(x) \geq 1 \\
 & \textbf{then } b(x) - 1 \textbf{ else } b(x) \textbf{ fi} + 1 \textbf{ else } \mathrm{without}^B(b,e')(x) \textbf{ fi fi} \\
= & \textbf{if } e = e' \textbf{ then } b \textbf{ else } \lambda x.\textbf{if } x = e \\
 & \textbf{then } \mathrm{without}^B(b,e')(x) + 1 \textbf{ else } \mathrm{without}^B(b,e')(x) \textbf{ fi fi} \\
= & \textbf{if } e = e' \textbf{ then } b \textbf{ else } \mathrm{nonemptybag}^B(e, \mathrm{without}^B(b,e')) \textbf{ fi}.
\end{array}
$$

The validity of the laws for elementsof follows immediately from the defining equations of the function. For minbag, we use the variant granted by Lemma A.5:

$$
\begin{array}{ll}
 & \mathrm{minbag}^B(\mathrm{nonemptybag}^B(e, \mathrm{emptybag}^B)) \\
= & \mathrm{minbag}^B(\mathrm{nonemptybag}^B(e, \lambda x.0)) \\
= & \mathrm{minbag}^B(\lambda x.\textbf{if } x = e \textbf{ then } (\lambda x.0)(x) + 1 \textbf{ else } (\lambda x.0)(x) \textbf{ fi}) \\
= & \mathrm{minbag}^B(\lambda x.\textbf{if } x = e \textbf{ then } 0 + 1 \textbf{ else } 0 \textbf{ fi}) \\
= & \min\{x \in \mathrm{elem} : \textbf{if } x = e \textbf{ then } 1 \textbf{ else } 0 \textbf{ fi} \geq 1\} \\
= & \min\{x \in \mathrm{elem} : \textbf{if } x = e \textbf{ then } 1 \geq 1 \textbf{ else } 0 \geq 1 \textbf{ fi}\} \\
= & \min\{x \in \mathrm{elem} : \textbf{if } x = e \textbf{ then } \mathrm{true} \textbf{ else } \mathrm{false} \textbf{ fi}\} \\
= & \min\{x \in \mathrm{elem} : x = e\} \\
= & \min\{e\} \\
= & e.
\end{array}
$$

$$\begin{aligned}
&\mathrm{minbag}^B(\mathrm{nonemptybag}^B(e, \mathrm{nonemptybag}^B(e', b)))\\
= \;&\mathrm{minbag}^B(\lambda x.\mathbf{if}\ x = e\ \mathbf{then}\ \mathrm{nonemptybag}^B(e', b)(x) + 1\\
&\mathbf{else}\ \mathrm{nonemptybag}^B(e', b)(x)\ \mathbf{fi})\\
= \;&\min\{x \in \mathrm{elem} : \mathbf{if}\ x = e\ \mathbf{then}\ \mathrm{nonemptybag}^B(e', b)(x) + 1\\
&\mathbf{else}\ \mathrm{nonemptybag}^B(e', b)(x)\ \mathbf{fi} \geq 1\}\\
= \;&\min\{x \in \mathrm{elem} : \mathbf{if}\ x = e\ \mathbf{then}\ \mathrm{nonemptybag}^B(e', b)(x) + 1 \geq 1\\
&\mathbf{else}\ \mathrm{nonemptybag}^B(e', b)(x) \geq 1\ \mathbf{fi}\}\\
= \;&\min\{x \in \mathrm{elem} : \mathbf{if}\ x = e\ \mathbf{then}\ \mathrm{true}\ \mathbf{else}\ \mathrm{nonemptybag}^B(e', b)(x) \geq 1\ \mathbf{fi}\}\\
= \;&\min\{x \in \mathrm{elem} : x = e \vee \mathrm{nonemptybag}^B(e', b)(x) \geq 1\}\\
= \;&\min(\{x \in \mathrm{elem} : x = e\} \cup \{x \in \mathrm{elem} : \mathrm{nonemptybag}^B(e', b)(x) \geq 1\})\\
= \;&\min(\{e\} \cup \{x \in \mathrm{elem} : \mathrm{nonemptybag}^B(e', b)(x) \geq 1\})\\
= \;&\min(e, \min\{x \in \mathrm{elem} : \mathrm{nonemptybag}^B(e', b)(x) \geq 1\})\\
= \;&\min(e, \mathrm{minbag}^B(\mathrm{nonemptybag}^B(e', b))).
\end{aligned}$$

□

Theorem A.10. *All models of the consistent specification Bag are behaviourally equivalent to the "classical" model of bags.*

Proof. Directly from Theorem A.8 and Lemma A.9. □

B Specification of the algebraic data type Tree

The specification of trees is contained in the theory file "tree.thy".

Theorem B.1. *In the specification of Tree, abstraction is surjective, even if its domain is restricted to heaps.*

Proof. Let b be an arbitrary element of the sort bag. By Lemma A.1, b can be generated by emptybag and nonemptybag, i.e., there exists a sequence $e_1, e_2, \ldots, e_k$ and a sequence $b_0, b_1, b_2, \ldots, b_k$ with

$$\begin{array}{lcll} b_0 & = & \text{emptybag} & \\ b_i & = & \text{nonemptybag}(e_i,\ b_{i-1}) & \quad (\text{for } 1 \le i \le k) \\ b_k & \equiv & \text{b} & \end{array}$$

For $k \in \mathbb{N}$ define $B_k =_{\text{def}} \{\text{b} \in \text{bag} \mid \exists e_1, \ldots, e_k, b_0, \ldots, b_k$ as just described$\}$. We show by induction that $\forall k \in \mathbb{N} : \forall \text{b} \in B_k : \exists \text{t} \in \text{Tree} : \text{heapinv(t)} \wedge \text{b} \equiv \text{abstraction(t)}$. This proves the Lemma, since bag$=\bigcup_{k \in \mathbb{N}} B_k$. If $k = 0$ then $B_k = \{\text{emptybag}\}$, heapinv(Leaf) = true, and abstraction(Leaf) = emptybag. For the induction step assume that $k \ge 1$ and $\forall j \in \{0, \ldots, k-1\} : \forall \text{b} \in B_j : \exists \text{t} \in \text{Tree} : \text{heapinv(t)} \wedge \text{b} \equiv \text{abstraction(t)}$. Let b $\in B_k$ and let $e_1, \ldots, e_k, b_0, \ldots, b_k$ be as described above.

First, we prove by induction that $\forall i \in \{1, \ldots, k\} : \text{minbag}(b_i) \in \{e_1, e_2, \ldots, e_i\}$. We rely on the variant specification granted by Lemma A.5. For $i = 1$, we calculate

$$\begin{array}{ll} & \text{minbag}(b_1) \\ = & \text{minbag}(\text{nonemptybag}(e_1,\ b_0)) \\ = & \text{minbag}(\text{nonemptybag}(e_1,\ \text{emptybag})) \\ \equiv & e_1 \\ \in & \{e_1\}. \end{array}$$

For the inductive step, assume that $i \in \{2, \ldots, k\}$ and $\text{minbag}(b_{i-1}) \in \{e_1, e_2, \ldots, e_{i-1}\}$. Then,

$$\begin{array}{ll} & \text{minbag}(b_i) \\ = & \text{minbag}(\text{nonemptybag}(e_i,\ b_{i-1})) \\ = & \text{minbag}(\text{nonemptybag}(e_i,\ \text{nonemptybag}(e_{i-1},\ b_{i-2}))) \\ \equiv & \min(e_i,\ \text{minbag}(\text{nonemptybag}(e_{i-1},\ b_{i-2}))) \\ = & \min(e_i,\ \text{minbag}(b_{i-1})) \\ \in & \{e_i\} \cup \{\text{minbag}(b_{i-1})\} \\ \subseteq & \{e_i\} \cup \{e_1, e_2, \ldots, e_{i-1}\} \\ = & \{e_1, e_2, \ldots, e_i\}. \end{array}$$

This finishes the inner induction, hence, minbag(b) $\equiv \text{minbag}(b_k) \in \{e_1, e_2, \ldots, e_k\}$. Let minbag(b) $= e_m$ for $1 \le m \le k$. Define

$$\begin{array}{lll} c_0 & =_{\text{def}} & \text{emptybag} \\ c_i & =_{\text{def}} & \text{nonemptybag}(e_{m+i},\ c_{i-1}) \quad (\text{for } 1 \le i \le k-m) \end{array}$$

We now have $c_{k-m} \in B_{k-m}$ and $b_{m-1} \in B_{m-1}$. By the induction hypothesis, there are heaps t_l, t_r such that $\text{abstraction}(t_l) \equiv c_{k-m}$ and $\text{abstraction}(t_r) \equiv b_{m-1}$. Define $t =_{\text{def}}$ Node t_l e_m t_r. We show that abstraction$(t) \equiv$ b.

To this end, we prove by induction that $\forall i \in \{0, \ldots, k-m\} : \text{unionbag}(c_i,\ b_m) \equiv b_{m+i}$. For $i = 0$, we calculate

$$\begin{array}{cl} & \text{unionbag}(c_0,\ b_m) \\ = & \text{unionbag}(\text{emptybag},\ b_m) \\ \equiv & b_m \\ = & b_{m+0}. \end{array}$$

For the inductive step, assume that $i \in \{1, \ldots, k-m\}$ and unionbag$(c_{i-1},\ b_m) \equiv b_{m+i-1}$. Then,

$$\begin{array}{cl} & \text{unionbag}(c_i,\ b_m) \\ = & \text{unionbag}(\text{nonemptybag}(e_{m+i},\ c_{i-1}),\ b_m) \\ \equiv & \text{nonemptybag}(e_{m+i},\ \text{unionbag}(c_{i-1},\ b_m)) \\ \equiv & \text{nonemptybag}(e_{m+i},\ b_{m+i-1}) \\ = & b_{m+i}. \end{array}$$

This finishes the inner induction, hence, unionbag$(c_{k-m},\ b_m) \equiv b_k$. Therefore,

$$\begin{array}{cl} & \text{abstraction}(t) \\ = & \text{abstraction}(\text{Node } t_l\ e_m\ t_r) \\ \equiv & \text{unionbag}(\text{abstraction}(t_l),\ \text{unionbag}(\text{singletonbag}(e_m),\ \text{abstraction}(t_r))) \\ \equiv & \text{unionbag}(c_{k-m},\ \text{unionbag}(\text{nonemptybag}(e_m,\ \text{emptybag}),\ b_{m-1})) \\ \equiv & \text{unionbag}(c_{k-m},\ \text{nonemptybag}(e_m,\ \text{unionbag}(\text{emptybag},\ b_{m-1}))) \\ \equiv & \text{unionbag}(c_{k-m},\ \text{nonemptybag}(e_m,\ b_{m-1})) \\ = & \text{unionbag}(c_{k-m},\ b_m) \\ \equiv & b_k \\ \equiv & \text{b}. \end{array}$$

This finishes the outer induction, hence, abstraction is surjective.

It remains to show heapinv(t), i.e., that t satisfies the heap property. By definition of the sequence $b_0, \ldots, b_k$ and an inductive argument using the recursive specification of minbag granted by Lemma A.5 we can prove $e_m = \min\{e_1, \ldots, e_k\}$. Since abstraction$(t_l) \equiv c_{k-m}$ and abstraction$(t_r) \equiv b_{m-1}$, the roots of the heaps t_l and t_r are $\in \{e_1, \ldots, e_k\}$. Therefore, in terms of the functions defined in the tree theory file, we have leroot$(e_m,\ t_l)$ and leroot$(e_m,\ t_r)$. Furthermore, we have heapinv(t_l) and heapinv(t_r) by the induction hypothesis. According to the definition of heapinv and that of t, we conclude that heapinv(t) holds. □

Theorem B.2. *In the specification of Tree,*

$$\forall t \in \mathit{Tree} : t = \mathit{Node}\ l\ e\ r \wedge \mathit{heapinv}(t) \quad \Rightarrow \quad \mathit{minbag}(\mathit{abstraction}(t)) \equiv e$$

and, hence, the rule tree_minimum is valid.

Proof. We proceed by structural induction over Tree. If t=Leaf, the implication holds vacuously. Otherwise, let t=Node l e r with heapinv(t) and, therefore, heapinv(l) and heapinv(r). If l=Leaf and r=Leaf, we calculate with Lemma A.5

$$\begin{aligned}
& \text{minbag(abstraction(t))} \\
= \; & \text{minbag(abstraction(Node Leaf e Leaf))} \\
= \; & \text{minbag(unionbag(unionbag(abstraction(Leaf), singletonbag(e)),} \\
& \quad \text{abstraction(Leaf)))} \\
= \; & \text{minbag(unionbag(unionbag(emptybag, nonemptybag(e, emptybag)),} \\
& \quad \text{emptybag))} \\
\equiv \; & \text{minbag(unionbag(nonemptybag(e, emptybag), emptybag))} \\
\equiv \; & \text{minbag(nonemptybag(e, unionbag(emptybag, emptybag)))} \\
\equiv \; & \text{minbag(nonemptybag(e, emptybag))} \\
\equiv \; & \text{e.}
\end{aligned}$$

If l=Leaf and r=Node l' e' r', we have e≤e' by the definition of heapinv. Let abstraction(r) ≡ nonemptybag(e', b') and calculate with Lemmas A.6 and A.5

$$\begin{aligned}
& \text{minbag(abstraction(t))} \\
= \; & \text{minbag(abstraction(Node Leaf e r))} \\
= \; & \text{minbag(unionbag(unionbag(abstraction(Leaf), singletonbag(e)),} \\
& \quad \text{abstraction(r)))} \\
\equiv \; & \text{minbag(unionbag(unionbag(emptybag, nonemptybag(e, emptybag)),} \\
& \quad \text{nonemptybag(e', b')))} \\
\equiv \; & \text{minbag(unionbag(nonemptybag(e, emptybag), nonemptybag(e', b')))} \\
\equiv \; & \text{min(minbag(nonemptybag(e, emptybag)), minbag(nonemptybag(e', b')))} \\
\equiv \; & \text{min(e, minbag(nonemptybag(e', b')))} \\
\equiv \; & \text{min(e, minbag(abstraction(r)))} \\
= \; & \text{min(e, minbag(abstraction(Node l' e' r')))} \\
\equiv \; & \text{min(e, e')} \\
= \; & \text{e,}
\end{aligned}$$

where the last two steps are due to the induction hypothesis and the definition of heapinv, respectively. If r=Leaf and l≠Leaf, we calculate with the associativity and commutativity of unionbag

$$\begin{aligned}
& \text{minbag(abstraction(t))} \\
= \; & \text{minbag(abstraction(Node l e Leaf))} \\
= \; & \text{minbag(unionbag(unionbag(abstraction(l), singletonbag(e)),} \\
& \quad \text{abstraction(Leaf)))} \\
\equiv \; & \text{minbag(unionbag(unionbag(abstraction(Leaf), singletonbag(e)),} \\
& \quad \text{abstraction(l)))} \\
= \; & \text{minbag(abstraction(Node Leaf e l))} \\
\equiv \; & \text{e,}
\end{aligned}$$

where the last step is due to the previous calculation. Finally, if l=Node l' e' r' and r=Node l" e" r", we have e≤e' and e≤e". Let abstraction(l) ≡ nonemptybag(e', b') and abstraction(r) ≡ nonemptybag(e", b") and calculate again with the associativity and commutativity of unionbag

$$
\begin{array}{rl}
 & \text{minbag(abstraction(t))} \\
= & \text{minbag(abstraction(Node l e r))} \\
= & \text{minbag(unionbag(unionbag(abstraction(l), singletonbag(e)),} \\
 & \quad \text{abstraction(r)))} \\
\equiv & \text{minbag(unionbag(singletonbag(e), unionbag(abstraction(l),} \\
 & \quad \text{abstraction(r))))} \\
\equiv & \text{minbag(unionbag(nonemptybag(e, emptybag),} \\
 & \quad \text{unionbag(nonemptybag(e', b'), abstraction(r))))} \\
\equiv & \text{minbag(unionbag(nonemptybag(e, emptybag),} \\
 & \quad \text{nonemptybag(e', unionbag(b', abstraction(r)))))} \\
\equiv & \text{min(minbag(nonemptybag(e, emptybag)),} \\
 & \quad \text{minbag(nonemptybag(e', unionbag(b', abstraction(r)))))} \\
\equiv & \text{min(e, minbag(nonemptybag(e', unionbag(b', abstraction(r)))))} \\
\equiv & \text{min(e, minbag(unionbag(nonemptybag(e', b'), nonemptybag(e'', b''))))} \\
\equiv & \text{min(e, min(minbag(nonemptybag(e', b')),} \\
 & \quad \text{minbag(nonemptybag(e'', b''))))} \\
\equiv & \text{min(e, min(minbag(abstraction(l)), minbag(abstraction(r))))} \\
\equiv & \text{min(e, min(e', e''))} \\
= & \text{e}
\end{array}
$$

using Lemmas A.6 and A.5, the induction hypothesis, and the definition of heapinv. □

Two transformation rules concerning heaps are stated in the theory files "tree-added.thy" (see Appendix D.8) and "sort-added.thy" (see Appendix D.11). Since they describe properties of the functions insert and removemin, respectively, they can only be proved after these functions have been introduced. Although in principle possible, a derivation of these rules is not well supported by *Ultra* due to the kind of reasoning needed (e.g., reasoning about arithmetic). Hence, we have decided to prove these rules manually. The first one is the transformation rule heap_insert.

Theorem B.3. *In the specification of Tree, for all trees l, r and elements e, x,*

$$
\begin{array}{lcl}
\textit{heapinv(Node l e r)} \wedge e \leq x & \Rightarrow & \textit{heapinv(Node (insert(x,r)) e l), and} \\
\textit{heapinv(Node l e r)} \wedge x \leq e & \Rightarrow & \textit{heapinv(Node (insert(e,r)) x l)}
\end{array}
$$

and, hence, the rule heap_insert is valid.

Proof. We will prove the two propositions in turn. Let l and r be trees, and let e and x be elements, such that heapinv(Node l e r) ∧ e ≤ x. By definition of heapinv we have heapinv(l) and leroot(e,l). We also have heapinv(r) and, therefore, by Theorem B.1, insert(x,r) is well-defined. In particular, heapinv(insert(x,r)) holds. It remains to show leroot(e,insert(x,r)). If r=Leaf, we calculate

$$
\begin{array}{rl}
 & \text{leroot(e,insert(x,r))} \\
= & \text{leroot(e,insert(x,Leaf))} \\
= & \text{leroot(e,Node Leaf x Leaf)} \\
= & \text{e} \leq \text{x},
\end{array}
$$

and that holds by assumption. Otherwise, let e' be the root of r and let e'' be the root of insert(x,r). Since leroot(e,r) holds, we have e ≤ e'. By definition of insert, we have equals(abstraction(insert(x,r)),nonemptybag(x,abstraction(r))).

We calculate with Theorem B.2, and Lemmas A.5 and A.4

$$\begin{array}{ll} & \text{e} \\ \leq & \text{min(x,e')} \\ \equiv & \text{min(x,minbag(abstraction(r)))} \\ \equiv & \text{minbag(nonemptybag(x,abstraction(r)))} \\ \equiv & \text{minbag(abstraction(insert(x,r)))} \\ \equiv & \text{e''}, \end{array}$$

and, hence, leroot(e,insert(x,r)).

For the second proposition, let x ≤ e. Again, by definition of heapinv we have heapinv(l) and leroot(e,l). From the latter, leroot(x,l) follows since x ≤ e. By the same argument as for the first proposition, heapinv(insert(e,r)) holds. It remains to show leroot(x,insert(e,r)). The calculations from the first proposition can be repeated in an analogous manner, using x ≤ e ≤ e' in the second case. □

The second transformation rule is heap_removemin_case.

Theorem B.4. *In the specification of Tree, for all trees l', l", r', r" and elements e, e', e",*

heapinv(Node (Node l' e' r') e (Node l" e" r")) ∧ e' ≤ e" ⇒
heapinv(Node (removemin(Node l' e' r')) e' (Node l" e" r")), and
heapinv(Node (Node l' e' r') e (Node l" e" r")) ∧ e" ≤ e' ⇒
heapinv(Node (Node l' e' r') e" (removemin(Node l" e" r")))

and, hence, the rule heap_removemin_case is valid.

Proof. We will prove the two propositions in turn. Let l, r, l', r', l", r" be trees and let e, e', e" be elements, such that l=Node l' e' r', r=Node l" e" r", and heapinv(Node l e r) ∧ e' ≤ e". By definition of heapinv we have heapinv(r). From e' ≤ e", leroot(e',r) follows. We also have heapinv(l) and, therefore, by Theorem B.1, removemin(l) is well-defined. In particular, heapinv(removemin(l)) holds. It remains to show leroot(e',removemin(l)). If l'=r'=Leaf, we calculate

$$\begin{array}{ll} & \text{leroot(e',removemin(l))} \\ = & \text{leroot(e',removemin(Node Leaf e' Leaf))} \\ = & \text{leroot(e',Leaf)} \\ = & \text{true.} \end{array}$$

Otherwise, let removemin(l)=Node l_0 e_0 r_0. By definition of removemin we have

equals(abstraction(removemin(l)),
without(abstraction(l),minbag(abstraction(l)))).

By definition of without and transitivity of equals we further get

equals(abstraction(removemin(l)),unionbag(abstraction(l'),abstraction(r'))).

If l'=Leaf and r'≠Leaf, let e_2 be the root of r' and calculate with Theorem B.2 and Lemma A.4

$$\begin{array}{ll} & \text{e'} \\ \leq & \text{e}_2 \\ \equiv & \text{minbag(abstraction(r'))} \\ \equiv & \text{minbag(unionbag(emptybag,abstraction(r')))} \\ = & \text{minbag(unionbag(abstraction(Leaf),abstraction(r')))} \\ = & \text{minbag(unionbag(abstraction(l'),abstraction(r')))} \\ \equiv & \text{minbag(abstraction(removemin(l)))} \\ \equiv & \text{e}_0, \end{array}$$

and, hence, we obtain leroot(e',removemin(l)). If l'≠Leaf and r'=Leaf, an analogous calculation can be performed. Finally, if l'≠Leaf and r'≠Leaf, let e_1 be the root of l' and let e_2 be the root of r' and calculate with Theorem B.2 and Lemma A.6

$$\begin{array}{ll} & \text{e'} \\ \leq & \text{min(e}_1\text{,e}_2\text{)} \\ \equiv & \text{min(minbag(abstraction(l')),minbag(abstraction(r')))} \\ \equiv & \text{minbag(unionbag(abstraction(l'),abstraction(r')))} \\ \equiv & \text{minbag(abstraction(removemin(l)))} \\ \equiv & \text{e}_0, \end{array}$$

and, therefore, leroot(e',removemin(l)) holds. The second proposition follows by observing that

heapinv(Node l e r) = heapinv(Node r e l).

□

C Dependances of the derived transformation rules

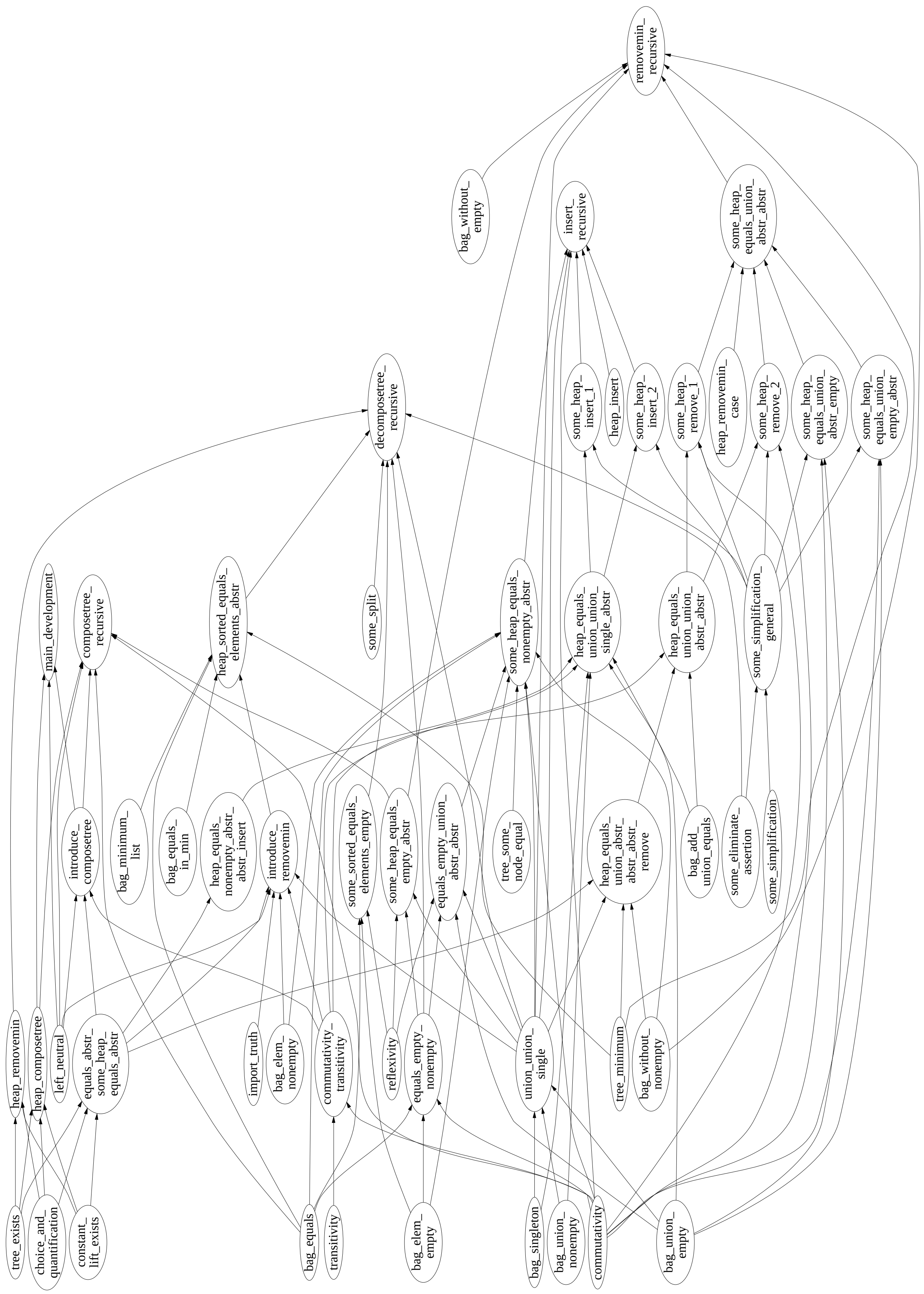
The following list describes the order in which the derivation of Heapsort proceeds. It shows the derived rules ("transform"), the applications of derived rules ("apply"), and the definitions of functions ("define"). Since *Ultra* does not support sub-derivations, they have to be performed by interrupting the main derivation (or in a bottom-up order for nice presentation). Each derived rule is complemented with a reference to the section of the Appendix that contains a description.

1. transform main_development (E.17)
   - 1.1. transform introduce_composetree (E.15)
     - 1.1.1. transform equals_abstr_some_heap_equals_abstr (E.4)
     - 1.1.2. transform commutativity_transitivity (E.1)
     - 1.1.3. apply equals_abstr_some_heap_equals_abstr
     - 1.1.4. define composetree
   - 1.2. transform heap_composetree (E.7)
   - 1.3. define decomposetree
   - 1.4. define heapsort
2. transform composetree_recursive (E.2)
   - 2.1. transform some_heap_equals_empty_abstr (E.19)
     - 2.1.1. transform union_union_single (E.30)
     - 2.1.2. transform equals_empty_nonempty (E.5)
   - 2.2. apply introduce_composetree
   - 2.3. apply heap_composetree
   - 2.4. define insert
3. transform insert_recursive (E.14)
   - 3.1. transform some_heap_equals_nonempty_abstr (E.20)
     - 3.1.1. apply union_union_single
     - 3.1.2. transform equals_empty_union_abstr_abstr (E.6)
       - 3.1.2.1. apply union_union_single
       - 3.1.2.2. apply equals_empty_nonempty
   - 3.2. apply union_union_single
   - 3.3. apply heap_insert
   - 3.4. transform some_heap_insert_1 (E.24)
     - 3.4.1. transform heap_equals_union_union_single_abstr (E.11)
       - 3.4.1.1. transform heap_equals_nonempty_abstr_abstr_insert (E.8)
         - 3.4.1.1.1. apply equals_abstr_some_heap_equals_abstr
       - 3.4.1.2. apply commutativity_transitivity
       - 3.4.1.3. apply heap_equals_nonempty_abstr_abstr_insert
     - 3.4.2. transform some_simplification_general (E.28)
   - 3.5. transform some_heap_insert_2 (E.25)
     - 3.5.1. apply heap_equals_union_union_single_abstr
     - 3.5.2. apply some_simplification_general
4. transform decomposetree_recursive (E.3)
   - 4.1. transform some_sorted_equals_elements_empty (E.29)
   - 4.2. apply union_union_single
   - 4.3. apply equals_empty_nonempty

- 4.4. transform heap_sorted_equals_elements_abstr (E.13)
  - 4.4.1. transform introduce_removemin (E.16)
    - 4.4.1.1. apply union_union_single
    - 4.4.1.2. apply equals_abstr_some_heap_equals_abstr
    - 4.4.1.3. apply commutativity_transitivity
    - 4.4.1.4. apply equals_abstr_some_heap_equals_abstr
    - 4.4.1.5. define removemin
- 4.5. transform heap_removemin (E.12)

5. transform removemin_recursive (E.18)
   - 5.1. apply some_heap_equals_empty_abstr
   - 5.2. apply union_union_single
   - 5.3. transform some_heap_equals_union_abstr_abstr (E.21)
     - 5.3.1. transform some_heap_equals_union_empty_abstr (E.23)
       - 5.3.1.1. apply some_simplification_general
     - 5.3.2. transform some_heap_equals_union_abstr_empty (E.22)
       - 5.3.2.1. apply some_simplification_general
     - 5.3.3. apply heap_removemin_case
     - 5.3.4. transform some_heap_remove_1 (E.26)
       - 5.3.4.1. transform heap_equals_union_union_abstr_abstr (E.10)
         - 5.3.4.1.1. transform heap_equals_union_abstr_abstr_abstr_remove (E.9)
           - 5.3.4.1.1.1. apply union_union_single
           - 5.3.4.1.1.2. apply equals_abstr_some_heap_equals_abstr
         - 5.3.4.1.2. apply commutativity_transitivity
         - 5.3.4.1.3. apply heap_equals_union_abstr_abstr_abstr_remove
       - 5.3.4.2. apply some_simplification_general
     - 5.3.5. transform some_heap_remove_2 (E.27)
       - 5.3.5.1. apply heap_equals_union_union_abstr_abstr
       - 5.3.5.2. apply some_simplification_general

The figure on the next page shows a graph that illustrates the dependances between the transformation rules. The nodes of the graph contain the names of the transformation rules that we use for the derivation of Heapsort. They include the user-defined rules and the derived rules but not the built-in rules. An edge from node v to node w in the graph means that the rule v is used (directly) during the derivation of the rule w.


removemin_ recursive
bag_without_ empty
insert_ recursive
some_heap_ equals_union_ abstr_abstr
decomposetree_ recursive
some_heap_ insert_1
heap_insert
some_heap_ insert_2
some_heap_ remove_1
heap_removemin_ case
some_heap_ remove_2
some_heap_ equals_union_ abstr_empty
some_heap_ equals_union_ empty_abstr
main_development
composetree_ recursive
heap_sorted_equals_ elements_abstr
some_split
some_heap_equals_ nonempty_abstr
heap_equals_ union_union_ single_abstr
heap_equals_ union_union_ abstr_abstr
some_simplification_ general
introduce_ composetree
bag_minimum_ list
bag_equals_ in_min
heap_equals_ nonempty_abstr_ abstr_insert
introduce_ removemin
some_sorted_equals_ elements_empty
some_heap_equals_ empty_abstr
equals_empty_union_ abstr_abstr
tree_some_ node_equal
heap_equals_ union_abstr_ abstr_abstr_ remove
bag_add_ union_equals
some_eliminate_ assertion
some_simplification
heap_removemin
heap_composetree
left_neutral
equals_abstr_ some_heap_ equals_abstr
import_truth
bag_elem_ nonempty
commutativity_ transitivity
reflexivity
equals_empty_ nonempty
union_union_ single
tree_minimum
bag_without_ nonempty
tree_exists
choice_and_ quantification
constant_ lift_exists
bag_equals
transitivity
bag_elem_ empty
bag_singleton
bag_union_ nonempty
commutativity
bag_union_ empty

We conclude with a table that shows which theory file each of the derived transformation rules belongs to. The theory files are presented in Appendix D.

transformation rule	theory file	section
commutativity_transitivity	general-derived.thy	D.3
composetree_recursive	tree-derived.thy	D.7
decomposetree_recursive	sort-derived.thy	D.10
equals_abstr_some_heap_equals_abstr	tree-derived.thy	D.7
equals_empty_nonempty	bag-derived.thy	D.5
equals_empty_union_abstr_abstr	tree-derived.thy	D.7
heap_composetree	tree-derived.thy	D.7
heap_equals_nonempty_abstr_abstr_insert	tree-derived.thy	D.7
heap_equals_union_abstr_abstr_abstr_remove	sort-derived.thy	D.10
heap_equals_union_union_abstr_abstr	sort-derived.thy	D.10
heap_equals_union_union_single_abstr	tree-derived.thy	D.7
heap_removemin	sort-derived.thy	D.10
heap_sorted_equals_elements_abstr	sort-derived.thy	D.10
insert_recursive	tree-derived.thy	D.7
introduce_composetree	tree-derived.thy	D.7
introduce_removemin	sort-derived.thy	D.10
main_development	sort-derived.thy	D.10
removemin_recursive	sort-derived.thy	D.10
some_heap_equals_empty_abstr	tree-derived.thy	D.7
some_heap_equals_nonempty_abstr	tree-derived.thy	D.7
some_heap_equals_union_abstr_abstr	sort-derived.thy	D.10
some_heap_equals_union_abstr_empty	tree-derived.thy	D.7
some_heap_equals_union_empty_abstr	tree-derived.thy	D.7
some_heap_insert_1	tree-derived.thy	D.7
some_heap_insert_2	tree-derived.thy	D.7
some_heap_remove_1	sort-derived.thy	D.10
some_heap_remove_2	sort-derived.thy	D.10
some_simplification_general	general-derived.thy	D.3
some_sorted_equals_elements_empty	sort-derived.thy	D.10
union_union_single	bag-derived.thy	D.5

D Source code

Appendix D.1 shows the *Ultra* project file for the derivation of the Heapsort. It lists the names of the theory files to be loaded. Data structures, functions, rules, and algebraic properties are logically organised into theories and physically stored in such theory files.

The source code of the specification is presented in Appendices D.2, D.4, D.6, and D.9. The results of the derivation appear in Appendices D.3, D.5, D.7, and D.10. Further parts of the specification that are manually included after parts of the derivation have been performed are given in Appendices D.8 and D.11. The final sorting algorithm is shown in Appendix D.12.

D.1 project

```
general.thy
general-derived.thy
bag.thy
bag-derived.thy
tree.thy
tree-derived.thy
tree-added.thy
sort.thy
sort-derived.thy
sort-added.thy
```

D.2 general.thy

```
-- ================================================================
-- General definitions extending the standard prelude.
-- ================================================================

-- ********
--! FUNS
-- ********

minimum :: [a] -> a
minimum [x] = x
minimum (x:y:zs) = x `min` (minimum (y:zs))

-- ********
--! RULES
-- ********

import_truth cn x =
  []
  |=
  x && cn True
  <=>
  x && cn x

-- ----------------------------------------------------------------
-- Algebra
-- ----------------------------------------------------------------
```

```
reflexivity r x =
  [ [] |- reflexive r ]
  |=
  r x x
  <=>
  True

transitivity r x y z =
  [ [] |- transitive r ]
  |=
  r x y && r y z
  <=>
  r x y && r x z

-- To apply commutativity for non-associative operations.
-- Symmetry is a special case of commutativity.

commutativity f x y =
  [ [] |- commutative f ]
  |=
  x `f` y
  <=>
  y `f` x

left_neutral f x y =
  [ [] |- lneutral f x ]
  |=
  y
  <=>
  x `f` y

-- ---------------------------------------------------------------
-- Non-deterministic expressions.
-- ---------------------------------------------------------------

-- The following rule is from problem 2-19 (b) (v) page 157 of [Man74].

constant_lift_exists p q =
  []
  |=
  exists (\x -> p && q x)
  <=>
  p && exists (\x -> q x)

-- The following rule is from appendix C.1.1 rule C.1.1 page 103 of [Sch98].
-- A more general version is proved in appendix D page 113 of [Sch98].

some_split p q =
  []
  |=
  some (\(x:xs) -> p x && q xs)
  <=>
  some p : some q
```

```
-- The following rule is from section 4.4.5.3 page 174 of [Par90].

choice_and_quantification f p =
  []
  |=
  some (\x -> exists (\y -> p y && f y == x))
  <=>
  f (some (\y -> p y))

-- The only rule to eliminate non-determinism.

some_simplification f p r y =
  [ [] |- reflexive r ]
  |=
  some (\x -> p x && p y && r (f x) (f y))
  ==>
  y

some_eliminate_assertion p q =
  []
  |=
  some (\x -> q && p x)
  ==>
  some (\x -> p x)

-- ********
--! CLAUSES
-- ********

associative (&&)
commutative (&&)
lneutral    (&&) True

commutative (==)
```

D.3 general-derived.thy

```
-- ********
--! FUNS
-- ********

-- ********
--! RULES
-- ********

commutativity_transitivity r x y z =
  [ [] |- commutative r , [] |- transitive r ]
  |=
  r x y && r z x
  <=>
  r x y && r z y
```

```
some_simplification_general f p q r y =
  [ [] |- reflexive r ]
  |=
  some (\x -> q && p x && p y && r (f x) (f y))
  ==>
  y

-- ********
--! CLAUSES
-- ********
```

D.4 bag.thy

```
-- ============================================================
-- Theory of Bags.
-- ============================================================

-- ********
--! FUNS
-- ********

-- Bag should be an ADT, not an algebraic DT.
-- By declaring the bag operations primitive, we cannot unfold them.

data Bag a = Bag

primitive emptybag     "emptybag"     :: Bag a
primitive nonemptybag  "nonemptybag"  :: a -> Bag a -> Bag a
primitive singletonbag "singletonbag" :: a -> Bag a
primitive unionbag     "unionbag"     :: Bag a -> Bag a -> Bag a
primitive without      "without"      :: Bag a -> a -> Bag a
primitive elembag      "elembag"      :: a -> Bag a -> Bool
primitive equals       "equals"       :: Bag a -> Bag a -> Bool
primitive minbag       "minbag"       :: Bag a -> a

-- Construction of a bag from a list.

elementsof :: [a] -> Bag a
elementsof [] = emptybag
elementsof (x:xs) = nonemptybag x (elementsof xs)

-- ********
--! RULES
-- ********

-- ------------------------------------------------------------
-- Constructive part of the specification.
-- ------------------------------------------------------------

-- singletonbag is a derived constructor.
```

```
bag_singleton e =
  []
  |=
  singletonbag e
  <=>
  nonemptybag e emptybag

-- unionbag is a derived constructor.

bag_union_empty b =
  []
  |=
  emptybag ‘unionbag‘ b
  <=>
  b

bag_union_nonempty e b1 b2 =
  []
  |=
  nonemptybag e b1 ‘unionbag‘ b2
  <=>
  nonemptybag e (b1 ‘unionbag‘ b2)

-- Axioms for without.

bag_without_empty e =
  []
  |=
  emptybag ‘without‘ e
  <=>
  emptybag

bag_without_nonempty b e1 e2 =
  []
  |=
  nonemptybag e1 b ‘without‘ e2
  <=>
  if e1 == e2 then b else nonemptybag e1 (b ‘without‘ e2)

-- Axioms for elembag.

bag_elem_empty e =
  []
  |=
  e ‘elembag‘ emptybag
  <=>
  False

bag_elem_nonempty b e1 e2 =
  []
  |=
  e1 ‘elembag‘ nonemptybag e2 b
  <=>
  e1 == e2 || e1 ‘elembag‘ b

-- The equality of two bags is sufficiently defined by the following rule.
-- The other cases are covered by reflexivity and commutativity.

bag_equals e b1 b2 =
  []
  |=
  nonemptybag e b1 ‘equals‘ b2
  <=>
  e ‘elembag‘ b2 && b1 ‘equals‘ (b2 ‘without‘ e)

-- ------------------------------------------------------------
-- Non-constructive part of the specification.
-- ------------------------------------------------------------

-- The minimum is specified according to construction from lists.
-- We have minbag . elementsof = minimum (where defined).

bag_minimum_list x xs =
  []
  |=
  minbag (elementsof (x:xs))
  <=>
  minimum (x:xs)

-- ------------------------------------------------------------
-- Observational equality.
-- ------------------------------------------------------------

bag_add_union_equals b1 b2 b3 =
  []
  |=
  b1 ‘equals‘ b2
  <=>
  (b3 ‘unionbag‘ b1) ‘equals‘ (b3 ‘unionbag‘ b2)

bag_equals_in_min b1 b2 cn =
  []
  |=
  b1 ‘equals‘ b2 && cn (minbag b1)
  <=>
  b1 ‘equals‘ b2 && cn (minbag b2)

-- ********
--! CLAUSES
-- ********

reflexive equals
transitive equals
commutative equals

associative unionbag
```

```
commutative unionbag
```

D.5 bag-derived.thy

```
-- ********
--! FUNS
-- ********

-- ********
--! RULES
-- ********

equals_empty_nonempty b e =
  []
  |=
  emptybag ‘equals‘ (nonemptybag e b)
  <=>
  False

union_union_single b1 b2 e =
  []
  |=
  b1 ‘unionbag‘ singletonbag e ‘unionbag‘ b2
  <=>
  nonemptybag e (b1 ‘unionbag‘ b2)

-- ********
--! CLAUSES
-- ********
```

D.6 tree.thy

```
-- ============================================================
-- Theory of binary, node-valued trees and heaps.
-- ============================================================

-- ********
--! FUNS
-- ********

data Tree a = Leaf | Node (Tree a) a (Tree a)

-- Binary trees are used as the implementation representation for bags.

abstraction :: Tree a -> Bag a
abstraction Leaf = emptybag
abstraction (Node l e r) =
  abstraction l ‘unionbag‘ singletonbag e ‘unionbag‘ abstraction r

-- The inductive predicate stating the heap property.
```

```
leroot :: a -> Tree a -> Bool
leroot x Leaf = True
leroot x (Node l e r) = x <= e

heapinv :: Tree a -> Bool
heapinv Leaf = True
heapinv (Node l e r) = heapinv l && heapinv r && leroot e l && leroot e r

-- ********
--! RULES
-- ********

-- -------------------------------------------------------------
-- Non-deterministic expressions.
-- -------------------------------------------------------------

-- The following rule is a special case of appendix D page 113 of [Sch98].

tree_some_node_equal x =
  []
  |=
  some (\(Node l e r) -> l == Leaf && e == x && r == Leaf)
  <=>
  Node Leaf x Leaf

-- Every bag has a representing heap.

tree_exists b =
  []
  |=
  exists (\t -> heapinv t && b `equals` abstraction t)
  <=>
  True

-- -------------------------------------------------------------
-- The heap property in action.
-- -------------------------------------------------------------

-- The smallest element of a bag is the root of its representation.

tree_minimum cn e l r =
  []
  |=
  heapinv (Node l e r) && cn (minbag (abstraction (Node l e r)))
  <=>
  heapinv (Node l e r) && cn e

-- ********
--! CLAUSES
-- ********
```

D.7 tree-derived.thy

```
-- ********
--! FUNS
-- ********

-- The function composetree is defined in step 6 of the derivation of
-- the rule introduce_composetree.

composetree :: [a] -> Tree a
composetree xs = some (\t -> heapinv t && elementsof xs ‘equals‘ abstraction t)

-- The function insert is defined in step 13 of the derivation of
-- the rule composetree_recursive.

insert :: a -> Tree a -> Tree a
insert x t = some (\s -> heapinv t && heapinv s &&
  nonemptybag x (abstraction t) ‘equals‘ abstraction s)

-- ********
--! RULES
-- ********

equals_abstr_some_heap_equals_abstr b =
  []
  |=
  b ‘equals‘ (abstraction (some (\t -> heapinv t && b ‘equals‘ abstraction t)))
  <=>
  True

introduce_composetree b xs =
  []
  |=
  b ‘equals‘ elementsof xs
  <=>
  b ‘equals‘ abstraction (composetree xs)

heap_composetree xs =
  []
  |=
  heapinv (composetree xs)
  <=>
  True

some_heap_equals_empty_abstr =
  []
  |=
  some (\t -> heapinv t && emptybag ‘equals‘ abstraction t)
  <=>
  Leaf

composetree_recursive xs =
  []
  |=
  composetree xs
  <=>
  case xs of
    [] -> Leaf
    y : ys -> insert y (composetree ys)

equals_empty_union_abstr_abstr l r =
  []
  |=
  emptybag `equals` (abstraction l `unionbag` abstraction r)
  <=>
  case l of
    Leaf -> case r of
              Leaf -> True
              Node d e f -> False
    Node b e c -> False

some_heap_equals_nonempty_abstr e =
  []
  |=
  some (\t -> heapinv t && nonemptybag e emptybag `equals` abstraction t)
  <=>
  Node Leaf e Leaf

heap_equals_nonempty_abstr_abstr_insert e t =
  []
  |=
  heapinv t && nonemptybag e (abstraction t) `equals` abstraction (insert e t)
  <=>
  heapinv t

heap_equals_union_union_single_abstr b1 b2 e t =
  []
  |=
  heapinv t && b1 `equals` (b2 `unionbag` (singletonbag e `unionbag` abstraction t))
  <=>
  heapinv t && b1 `equals` (b2 `unionbag` abstraction (insert e t))

some_heap_insert_1 e l r x =
  []
  |=
  some (\s -> (heapinv (Node l e r) && heapinv (Node (insert x r) e l)) &&
               heapinv s && abstraction s `equals` (abstraction l `unionbag`
               singletonbag e `unionbag` singletonbag x `unionbag` abstraction r))
  ==>
  Node (insert x r) e l

some_heap_insert_2 e l r x =
  []
  |=
  some (\s -> (heapinv (Node l e r) && heapinv (Node (insert e r) x l)) &&
               heapinv s && abstraction s `equals` (abstraction l `unionbag`
               singletonbag e ‘unionbag‘ singletonbag x ‘unionbag‘ abstraction r))
  ==>
  Node (insert e r) x l

insert_recursive x t =
  []
  |=
  insert x t
  ==>
  case t of
    Leaf -> Node Leaf x Leaf
    Node l e r -> if e <= x then Node (insert x r) e l
                            else Node (insert e r) x l

some_heap_equals_union_empty_abstr e r =
  []
  |=
  some (\t -> heapinv (Node Leaf e r) && heapinv t &&
    (abstraction Leaf ‘unionbag‘ abstraction r) ‘equals‘ abstraction t)
  ==>
  r

some_heap_equals_union_abstr_empty e1 e2 l r =
  []
  |=
  some (\t -> heapinv (Node (Node l e1 r) e2 Leaf) && heapinv t &&
    (abstraction (Node l e1 r) ‘unionbag‘ abstraction Leaf) ‘equals‘ abstraction t)
  ==>
  Node l e1 r

-- ********
--! CLAUSES
-- ********
```

D.8 tree-added.thy

```
-- ********
--! FUNS
-- ********

-- ********
--! RULES
-- ********

heap_insert e l r x =
  []
  |=
  heapinv (Node l e r)
  <=>
  heapinv (Node l e r) && if e <= x then heapinv (Node (insert x r) e l)
                                    else heapinv (Node (insert e r) x l)
```

```
-- ********
--! CLAUSES
-- ********
```

D.9 sort.thy

```
-- ================================================================
-- Theory for the sorting problem.
-- ================================================================

-- ********
--! FUNS
-- ********

-- ----------------------------------------------------------------
-- The traditional specification of the sorting algorithm.
-- ----------------------------------------------------------------

sort :: [a] -> [a]
sort xs = some (\ys -> issorted ys && elementsof ys `equals` elementsof xs)

issorted :: [a] -> Bool
issorted [] = True
issorted (x:xs) = x == minimum (x:xs) && issorted xs

-- ********
--! RULES
-- ********

-- ********
--! CLAUSES
-- ********
```

D.10 sort-derived.thy

```
-- ********
--! FUNS
-- ********

-- The function decomposetree is defined in step 6 of the derivation of
-- the rule main_development.

decomposetree :: Tree a -> [a]
decomposetree t = some (\xs -> heapinv t && issorted xs &&
  elementsof xs `equals` abstraction t)

-- The function heapsort is defined in step 7 of the derivation of
-- the rule main_development.

heapsort :: [a] -> [a]
heapsort xs = decomposetree (composetree xs)
```

```
-- The function removemin is defined in step 11 of the derivation of
-- the rule introduce_removemin.

removemin :: Tree a -> Tree a
removemin t = some (\s -> heapinv t && heapinv s &&
  (abstraction t `without` minbag (abstraction t)) `equals` abstraction s)

-- ********
--! RULES
-- ********

main_development xs =
  []
  |=
  sort xs
  <=>
  heapsort xs

some_sorted_equals_elements_empty =
  []
  |=
  some (\xs -> issorted xs && elementsof xs `equals` emptybag)
  <=>
  []

introduce_removemin e l r xs =
  []
  |=
  heapinv (Node l e r) && e `elembag`
    (abstraction l `unionbag` singletonbag e `unionbag` abstraction r) &&
    elementsof xs `equals`
    (abstraction (Node l e r) `without` minbag (abstraction (Node l e r)))
  <=>
  heapinv (Node l e r) &&
  elementsof xs `equals` abstraction (removemin (Node l e r))

heap_removemin e l r =
  []
  |=
  heapinv (Node l e r) && heapinv (removemin (Node l e r))
  <=>
  heapinv (Node l e r)

heap_sorted_equals_elements_abstr e l r x xs =
  []
  |=
  heapinv (Node l e r) && issorted (x : xs) &&
    elementsof (x : xs) `equals` abstraction (Node l e r)
  <=>
  x == e && heapinv (Node l e r) && issorted xs &&
    elementsof xs `equals` abstraction (removemin (Node l e r))
```

```
decomposetree_recursive t =
  []
  |=
  decomposetree t
  ==>
  case t of
    Leaf -> []
    Node l e r -> e : decomposetree (removemin (Node l e r))

heap_equals_union_abstr_abstr_abstr_remove e l r =
  []
  |=
  heapinv (Node l e r) && (abstraction l ‘unionbag‘ abstraction r) ‘equals‘
    (abstraction (removemin (Node l e r)))
  <=>
  heapinv (Node l e r)

heap_equals_union_union_abstr_abstr b1 b2 e l r =
  []
  |=
  heapinv (Node l e r) && b1 ‘equals‘
    (b2 ‘unionbag‘ (abstraction l ‘unionbag‘ abstraction r))
  <=>
  heapinv (Node l e r) && b1 ‘equals‘
    (b2 ‘unionbag‘ abstraction (removemin (Node l e r)))

some_heap_remove_1 e e1 e2 l1 l2 r1 r2 =
  []
  |=
  some (\t -> (heapinv (Node (Node l1 e1 r1) e (Node l2 e2 r2)) &&
    heapinv (Node (removemin (Node l1 e1 r1)) e1 (Node l2 e2 r2))) && heapinv t &&
    (abstraction (Node l1 e1 r1) ‘unionbag‘ abstraction (Node l2 e2 r2)) ‘equals‘
    abstraction t)
  ==>
  Node (removemin (Node l1 e1 r1)) e1 (Node l2 e2 r2)

some_heap_remove_2 e e1 e2 l1 l2 r1 r2 =
  []
  |=
  some (\t -> (heapinv (Node (Node l1 e1 r1) e (Node l2 e2 r2)) &&
    heapinv (Node (Node l1 e1 r1) e2 (removemin (Node l2 e2 r2)))) && heapinv t &&
    (abstraction (Node l1 e1 r1) ‘unionbag‘ abstraction (Node l2 e2 r2)) ‘equals‘
    abstraction t)
  ==>
  Node (Node l1 e1 r1) e2 (removemin (Node l2 e2 r2))

some_heap_equals_union_abstr_abstr l e r =
  []
  |=
  some (\t -> heapinv (Node l e r) && heapinv t &&
    (abstraction l ‘unionbag‘ abstraction r) ‘equals‘ abstraction t)
  ==>
  case l of
```

```
    Leaf -> r
    Node l1 e1 r1 ->
      case r of
        Leaf -> Node l1 e1 r1
        Node l2 e2 r2 ->
          if e1 <= e2 then Node (removemin (Node l1 e1 r1)) e1 (Node l2 e2 r2)
                      else Node (Node l1 e1 r1) e2 (removemin (Node l2 e2 r2))

removemin_recursive t =
  []
  |=
  removemin t
  ==>
  case t of
    Leaf -> Leaf
    Node l e r ->
      case l of
        Leaf -> r
        Node l1 e1 r1 ->
          case r of
            Leaf -> Node l1 e1 r1
            Node l2 e2 r2 ->
              if e1 <= e2 then Node (removemin (Node l1 e1 r1)) e1 (Node l2 e2 r2)
                          else Node (Node l1 e1 r1) e2 (removemin (Node l2 e2 r2))

-- ********
--! CLAUSES
-- ********
```

D.11 sort-added.thy

```
-- ********
--! FUNS
-- ********

-- ********
--! RULES
-- ********

heap_removemin_case e e1 e2 l1 l2 r1 r2 =
  []
  |=
  heapinv (Node (Node l1 e1 r1) e (Node l2 e2 r2))
  <=>
  heapinv (Node (Node l1 e1 r1) e (Node l2 e2 r2)) &&
    if e1 <= e2 then heapinv (Node (removemin (Node l1 e1 r1)) e1 (Node l2 e2 r2))
                else heapinv (Node (Node l1 e1 r1) e2 (removemin (Node l2 e2 r2)))

-- ********
--! CLAUSES
-- ********
```

D.12 heapsort.hs

```
heapsort :: Ord a => [a] -> [a]
heapsort xs = decomposetree (composetree xs)

data Tree a = Leaf | Node (Tree a) a (Tree a)

composetree :: Ord a => [a] -> Tree a
composetree xs =
  case xs of
    [] -> Leaf
    y : ys -> insert y (composetree ys)

insert :: Ord a => a -> Tree a -> Tree a
insert x t =
  case t of
    Leaf -> Node Leaf x Leaf
    Node l e r -> if e <= x then Node (insert x r) e l
                            else Node (insert e r) x l

decomposetree :: Ord a => Tree a -> [a]
decomposetree t =
  case t of
    Leaf -> []
    Node l e r -> e : decomposetree (removemin (Node l e r))

removemin :: Ord a => Tree a -> Tree a
removemin t =
  case t of
    Leaf -> Leaf
    Node l e r ->
      case l of
        Leaf -> r
        Node l1 e1 r1 ->
          case r of
            Leaf -> Node l1 e1 r1
            Node l2 e2 r2 ->
              if e1 <= e2 then Node (removemin (Node l1 e1 r1)) e1 (Node l2 e2 r2)
                          else Node (Node l1 e1 r1) e2 (removemin (Node l2 e2 r2))
```

E Description of the derived transformation rules

We describe the derivations of all transformation rules. The derivation protocols are available at http://www.informatik.uni-ulm.de/pm/mitarbeiter/walter/data/heapsort/.

E.1 commutativity_transitivity

This rule describes a special way to apply the transitivity rule in the presence of a commutative (i.e., symmetric) relation. The reason for deriving this rule is that it is used several times. It is one of two derived rules that have applicability conditions.

E.2 composetree_recursive

This rule describes the derivation of a recursive version for the descriptive specification of the function composetree. This function is used to build a heap that consists of the elements of the parameter list. The derivation proceeds according to the unfold-fold methodology. First, the special case of the empty list is treated. In the other case, one element is isolated and the remaining list is transformed recursively into a tree. Afterwards, the isolated element is inserted into the already constructed tree with the help of an auxiliary function, namely insert, that has a descriptive specification.

E.3 decomposetree_recursive

This rule describes the derivation of a recursive version for the descriptive specification of the function decomposetree. This function is used to extract a sorted list from a heap. The derivation proceeds according to the unfold-fold methodology. First, the special case of the empty heap is treated. If the heap is not empty, we rule out the case that the resulting list is empty. In the other case, the root of the heap, which is its minimum element, is identified with the head of the resulting list, that must also be the minimum element. The root of the heap is removed with the help of an auxiliary function, namely removemin, that has a descriptive specification. That function returns a heap from which the remaining list is extracted recursively.

E.4 equals_abstr_some_heap_equals_abstr

This rule describes the fact that every bag may be written as the abstraction of a representing heap. From the specification of trees we already know that for every bag there exists a representing heap. This existence proposition is turned into a constructive specification with this rule. It is conceivable to state such a basic rule directly in the specification of the problem, but we have chosen to derive it from an even more basic rule, namely choice_and_quantification.

E.5 equals_empty_nonempty

This rule derives a special case of the equality of two bags that is used several times in the derivation of Heapsort.

E.6 equals_empty_union_abstr_abstr

This rule describes that the only way to write the empty bag as the union of two representing trees is to have both trees empty. Its derivation proceeds by analysing all cases.

E.7 heap_composetree

This rule describes the fact that every tree that results from the application of composetree is a heap. The derivation of this rule parallels that of equals_abstr_some_heap_equals_abstr.

E.8 heap_equals_nonempty_abstr_abstr_insert

This rule describes the effect of the function insert when applied to an element and a heap. The resulting tree represents the addition of the element to the collection of elements represented by the heap. The rule is a direct consequence of the specification of insert.

E.9 heap_equals_union_abstr_abstr_abstr_remove

This rule describes the effect of the function removemin when applied to a heap. The resulting tree represents the union of the collections of elements represented by the two sub-trees. The rule is a consequence of the specification of removemin and the fact that the minimum element of a heap is its root.

E.10 heap_equals_union_union_abstr_abstr

This rule lifts the effect of the function removemin into a certain context. The context is given by first uniting with a bag and then comparing for equality to another bag. This rule is needed because the equality on bags describes behavioural equivalence rather than structural equivalence.

E.11 heap_equals_union_union_single_abstr

This rule lifts the effect of the function insert into a certain context. The context is given by first uniting with a bag and then comparing for equality to another bag. This rule is needed because the equality on bags describes behavioural equivalence rather than structural equivalence.

E.12 heap_removemin

This rule describes the fact that every tree that results from applying removemin to a heap is a heap. The derivation of this rule parallels that of equals_abstr_some_heap_equals_abstr.

E.13 heap_sorted_equals_elements_abstr

This rule transforms the relationship between a non-empty sorted list and a heap that represent the same bag. This relationship can be expressed as follows. The head of the list is equal to the root of the heap and the tail of the list represents the same bag as the heap after its root has been removed. We employ the knowledge from the specification that the minimum element is located at the head of the sorted list, and at the root of the heap, respectively. The function removemin that is defined in another rule is used to describe the remaining list.

E.14 insert_recursive

This rule describes the derivation of a recursive version for the descriptive specification of the function insert. This function is used to insert an element into a heap. The derivation proceeds according to the unfold-fold methodology. First, the special case of the empty heap is treated. Otherwise, we distinguish two cases: either the element to be inserted is less than the root of the heap, or vice versa. In both cases the smaller element will be the new root while the greater element is recursively inserted into the right sub-tree. Both sub-trees will be swapped finally to guarantee that Braun trees with logarithmic access time are constructed.

E.15 introduce_composetree

This rule introduces the function composetree that constructs the intermediate heap from the input sequence. We use the fact that the bag containing the collection of elements of the input sequence has a representing heap (like any other bag). The function composetree is defined to construct such a heap.

E.16 introduce_removemin

This rule introduces the function removemin that removes the minimum element of a heap. We use the fact that the bag containing the collection of elements of the heap without its minimum element has a representing heap (like any other bag). The function removemin is defined to construct such a heap.

E.17 main_development

The main development of Heapsort consists of the introduction of an intermediate heap and the extraction of the sorted list from that heap. The construction of the heap, namely composetree, is defined in the derivation of another rule. The extraction, namely decomposetree, is defined in the derivation of this rule to do exactly the remaining work. The function heapsort is then defined as the function composition of composetree and decomposetree.

E.18 removemin_recursive

This rule describes the derivation of a recursive version for the descriptive specification of the function removemin. This function is used to remove the minimum element from a heap. The derivation proceeds according to the unfold-fold methodology. First, the special case of the empty heap is treated. In the other case, the root of the heap, which is its minimum element, is removed. The remaining entity represents the union of the collections of elements of both sub-trees. An auxiliary rule is used to construct some heap that represents this collection.

E.19 some_heap_equals_empty_abstr

This rule treats the first case of the derivation of a recursive version for the functions composetree and removemin. The only heap that represents the empty list is the empty heap. The derivation proceeds by case-analysis and elimination of the non-empty heap as a candidate.

E.20 some_heap_equals_nonempty_abstr

This rule treats the first case of the derivation of a recursive version for the function insert. An element is to be inserted into the empty heap. The only heap that represents the bag containing one element is the heap with that element at its root and empty sub-trees. To derive this, the empty heap is eliminated as a candidate, first. Then, the root is identified to be the element that is inserted. By case-analysis the sub-trees are identified to be empty.

E.21 some_heap_equals_union_abstr_abstr

This rule merges two heaps into a single heap. This operation is used after removing the minimum element of a heap. The rule constructs some heap that represents a bag that is the union of the collections represented by two heaps. Let us call these two heaps the left heap and the right heap. The derivation proceeds by case-analysis on the left heap and the right heap. Either one of these heaps may be empty or not empty. If the left heap is empty, the result is the right heap. If the right heap is empty, the result is the left heap. If both heaps are not empty, we distinguish two cases: either the root of the right heap is less than the root of the left heap, or vice versa. In both cases the smaller root will be the new root and it is recursively removed from the heap it belongs to. Altogether, there are four cases, and all of them are treated by auxiliary rules.

E.22 some_heap_equals_union_abstr_empty

This rule treats the case of merging two heaps where the right heap is empty, but not the left one. The union of the two bags that are represented by the left heap and the right heap is reduced to the bag that is represented by the left heap. Therefore, the left heap solves this problem.

E.23 some_heap_equals_union_empty_abstr

This rule treats the case of merging two heaps where the left heap is empty. The union of the two bags that are represented by the left heap and the right heap is reduced to the bag that is represented by the right heap. Therefore, the right heap solves this problem.

E.24 some_heap_insert_1

This rule treats the case of inserting an element into a heap where the heap is not empty and the root of the heap is less than or equal to the element to be inserted. Then, the bags represented by the heap and the single element are rearranged with the help of associativity and commutativity. The root of the heap becomes the new root and the element is recursively inserted into the right sub-tree. The right and the left sub-trees are swapped to ensure the construction of Braun trees. Note that there are many ways to rearrange the elements into a new heap. This is reflected by the fact that this rule denotes a descendance between its input and output program scheme, rather than the equivalence relation.

E.25 some_heap_insert_2

This rule treats the case of inserting an element into a heap where the heap is not empty and the element to be inserted is less than the root of the heap. Then, the bags represented by the heap and the single element are rearranged with the help of associativity and commutativity. The element to be inserted becomes the new root and the root of the heap is recursively inserted into the right sub-tree. The right and the left sub-trees are swapped to ensure the construction of Braun trees. Note that there are many ways to rearrange the elements into a new heap. This is reflected by the fact that this rule denotes a descendance between its input and output program scheme, rather than the equivalence relation.

E.26 some_heap_remove_1

This rule treats the case of merging two heaps where both heaps are not empty and the root of the left heap is less than or equal to the root of the right heap. Then, the bags represented by the heaps are rearranged with the help of associativity and commutativity. The root of the left heap becomes the new root and is recursively removed from the left heap. Note that there are many ways to rearrange the remaining elements into a new heap. This is reflected by the fact that this rule denotes a descendance between its input and output program scheme, rather than the equivalence relation.

E.27 some_heap_remove_2

This rule treats the case of merging two heaps where both heaps are not empty and the root of the right heap is less than the root of the left heap. Then, the bags represented by the heaps are rearranged with the help of associativity and commutativity. The root of the right heap becomes the new root and is recursively removed from the right heap. Note that there are many ways to rearrange the remaining elements into a new heap. This is reflected by the fact that this rule denotes a descendance between its input and output program scheme, rather than the equivalence relation.

E.28 some_simplification_general

This rule describes a more general version of the rule some_simplification that is specified in "general.thy". Since we assume that all expressions are defined, the boolean value q is necessarily true in the input scheme. It is therefore eliminated from the conjunction and the original simplification rule can be applied. This rule is one of two derived rules that have applicability conditions.

E.29 some_sorted_equals_elements_empty

This rule treats the first case of the derivation of a recursive version for the function decomposetree. The only sorted list that is represented by the empty heap is the empty list. The derivation proceeds by case-analysis and elimination of the non-empty list as a candidate.

E.30 union_union_single

This rule rearranges the union of two bags and a one-element bag in a way that is used several times in the derivation of Heapsort.

References

[DF88] E.W. Dijkstra and W.H.J. Feijen. *A Method of Programming.* Addison-Wesley Publishing Company, 1988.

[Gut00] W.N. Guttmann. *An Introduction to Ultra.* University of Ulm, December 2000. `http://www.informatik.uni-ulm.de/pm/ultra/`.

[Man74] Z. Manna. *Mathematical Theory of Computation.* McGraw-Hill, 1974.

[ML97] S.M. Merritt and K.-K. Lau. A logical inverted taxonomy of sorting algorithms. In S. Kuru, M.U. Caglayan, and H.L. Akin, editors, *Proceedings of the Twelfth International Symposium on Computer and Information Sciences*, pages 576–583. Bogazici University, 1997. `http://www.cs.man.ac.uk/~kung-kiu/pub/iscis97.ps.gz`.

[Par83] H. Partsch. An exercise in the transformational derivation of an efficient program by joint development of control and data structure. *Science of Computer Programming*, 3(1):1–35, April 1983.

[Par90] H.A. Partsch. *Specification and Transformation of Programs: A Formal Approach to Software Development.* Springer-Verlag, 1990.

[Pat92] R. Paterson. A tiny functional language with logical features. In J. Darlington and R. Dietrich, editors, *Declarative Programming*, pages 66–79. Springer-Verlag, 1992. Phoenix Seminar and Workshop on Declarative Programming, Sasbachwalden, Black Forest, Germany, 18-22 November 1991.

[Pep87] P. Pepper. A simple calculus for program transformation (inclusive of induction). *Science of Computer Programming*, 9(3):221–262, December 1987.

[Sch98] J. Schmid. Nichtdeterminismus und Programmtransformation. Master's thesis, University of Ulm, 1998.

[SS92] H. Søndergaard and P. Sestoft. Non-determinism in functional languages. *The Computer Journal*, 35(5):514–523, October 1992.

[Wir90] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 13, pages 675–788. Elsevier Science Publishers, 1990.

Contents