Deriving Sorting Algorithms

José Bacelar Almeida and Jorge Sousa Pinto
Departamento de Informática
Universidade do Minho
4710-057 Braga, Portugal
{jba,jsp}@di.uminho.pt

Techn. Report DI-PURe-06.04.01
2006, April

PURe
Program Understanding and Re-engineering: Calculi and Applications
(Project POSI/ICHS/44304/2002)

Departamento de Informática da Universidade do Minho
Campus de Gualtar — Braga — Portugal

DI-PURe-06.04.01
Deriving Sorting Algorithms by José Bacelar Almeida and Jorge Sousa Pinto

Abstract

This paper shows how 3 well-known sorting algorithms can be derived by similar sequences of transformation steps from a common specification. Each derivation uses an auxiliary algorithm based on insertion into an intermediate structure. The proofs given involve both inductive and coinductive reasoning, which are here expressed in the same program calculation framework, based on unicity properties.

Deriving Sorting Algorithms

José Bacelar Almeida and Jorge Sousa Pinto
{jba,jsp}@di.uminho.pt

Departamento de Informática
Universidade do Minho
4710-057 Braga, Portugal

Abstract. This paper shows how 3 well-known sorting algorithms can be derived by similar sequences of transformation steps from a common specification. Each derivation uses an auxiliary algorithm based on insertion into an intermediate structure. The proofs given involve both inductive and coinductive reasoning, which are here expressed in the same *program calculation* framework, based on unicity properties.

1 Introduction

This paper is a case study on the derivation of correct functional algorithms from a specification, by program transformation. What this means is that the specification is simultaneously a description of the problem and also an initial, not very efficient solution to it, from which other more efficient solutions are derived.

The programs studied implement 3 classic divide-and-conquer sorting algorithms, and the proofs of correctness of the transformations are based on the view of these algorithms as hylomorphisms [1], i.e., as programs whose definition consists of the composition of a recursive and a co-recursive component. Proofs will involve a coinductive as well as an inductive argument, which will in general both be presented as resulting from the application of a unicity property.

Approaches to proving sorting algorithms correct usually involve a formalization of the "is sorted" property which is avoided in this paper. Our approach also allows us to prove inductively certain properties of the intermediate data-structures, which are representations of the recursion trees of the sorting algorithms. The properties include *well-balancing* properties which directly affect the efficiency of the algorithms, and also *structural* properties: for instance, as expected, the intermediate data-structure used by heapsort is a heap. Such properties cannot be proved for the divide-and-conquer algorithms using induction, since these data-structures are constructed in a co-recursive manner, but the algorithms given in this paper construct exactly the same structures using a fold over the argument list.

The paper is structured as follows: Section 2 reviews standard material on sorting in the functional setting, including the 3 algorithms that will be considered in the main sections of the paper. Section 3 contains background material on program calculation, based on unicity properties of recursion pattern operators. Section 4 then introduces two generic algorithms for sorting, based on insertion into an intermediate structure of a container type (in a leftwards and rightwards fashion respectively). Sections 5, 6, and 7 present the derivations of merge sort, quicksort, and heapsort, which are based on instantiations of the generic algorithms. Finally we conclude the paper in Section 8.

2 Sorting Homomorphisms and Divide-and-conquer Algorithms

Consider the simplest algorithm for sorting a list, usually known under the name of *insertion sort*. We give it here written in Haskell.

```
isort []     = []
isort (x:xs) = insert x (isort xs)
```

where `insert` inserts an element in a sorted list. This is certainly the most natural way of sorting a list in a traditional functional language: since the list structurally consists of a head element `x` and a tail sublist `xs`, it is natural to recursively sort `xs` and then combine this sorted list with `x`. This pattern of recursion can be captured by the foldr operator, resulting in the following definition where explicit recursion has been removed.

```
isort = foldr insert []
```

Actually, any sorting function is a *list homomorphism* [3, 6], which means that if the initial unsorted list is split at any point and the two resulting sublists are recursively sorted, there exists a binary operator $\odot$ that can combine the two results to give the final sorted list.

$$\mathsf{isort}\ (l_1 +\!\!+ l_2) = (\mathsf{isort}\ l_1) \odot (\mathsf{isort}\ l_2)$$

This $\odot$ operator is of course the (linear time) function of type $[a] \to [a] \to [a]$ that merges two sorted lists. A consequence of this (by Bird's Specialization Theorem) is that lists can also be sorted rightwards using the foldl operator:

```
isort = foldl insert' []
        where insert' l x = insert x l
```

In fact, isort is the unique homomorphism that can be calculated in these two ways.

The operator $\odot$ is associative with the empty list as unit, forming a monoid over lists. It is also commutative. insert and $\odot$ are related as follows

$$\mathsf{insert}\ x\ l = [x] \odot l \tag{1}$$

Insertion sort runs in quadratic time. Most well-know efficient sorting algorithms perform recursion *twice*, on subsequences obtained from the input sequence, and then combine the results (for this reason they are called *divide-and-conquer* algorithms). As such, they do not fit the simple iteration pattern captured by foldr. In the following we describe three different divide-and-conquer algorithms.

Heapsort. The principle behind heapsort is to traverse the list to obtain, in linear time, its minimum element y and a pair of lists of approximately equal size, containing the remaining elements (function haux). The lists are then recursively sorted and merged together, and y pasted at the head of the resulting list.

```
haux x [] = (x,[],[])
haux x (y:ys) = let (z,l,r) = haux y ys
                in if x<z then (x,z:r,l) else (z,x:r,l)

hsort [] = []
hsort xs = let (y,l,r) = haux xs
           in y:(merge (hsort l) (hsort r))
```

Quicksort. The criterion for obtaining the two sublists is here to use the head of the list (the only element accessible in constant time) as a pivot used to separate the remaining elements. The two sorted results need only be concatenated (with the pivot in the middle) to give the final result.

```
qaux _ []    = ([],[])
qaux x (h:t) = let (l,r) = qaux x t
               in if h<=x then (h:l,r) else (l,h:r)

qsort []     = []
qsort (x:xs) = let (l,r) = qaux x xs
               in (qsort l) ++ x:(qsort r)
```

Merge Sort. This is similar to heapsort except that the minimum element is not extracted when the list is traversed. For this reason an extra base case is used.

```
maux []     = ([],[])
maux (x:xs) = (x:b,a)  where (a,b) = maux xs

msort []  = []
msort [x] = [x]
msort xs  = let (l,r) = maux xs
            in merge (msort l) (msort r)
```

These functional versions of the algorithms may be difficult to recognize for a reader used to the imperative formulations, where the sorting is usually done in place, on indexed arrays. All three are however widely known in the functional programming community formulated as above.

3 Recursion Patterns, Unicity, and Hylomorphisms

The fold recursion pattern can be generalized for any regular type; in the context of the algebraic theory of data-types folds are *datatype-generic* (in the sense that they are parameterized by the base functor of the type), and usually called *catamorphisms*. The result of a fold on a node of some tree data-type is a combination of the results of recursively processing each subtree (and the contents of the node, if not empty).

The dual notion is the *unfold* (also called *anamorphism*): a function that constructs (possibly infinite) trees in the most natural way, in the sense that the subtrees of a node are recursively constructed by unfolding.

In the present paper we will need to work with two flavours of binary trees: leaf-labelled (for merge sort) and node-labelled trees (for the remaining algorithms). These types, and the corresponding recursion patterns, are defined in Table 1.

In principle, a fold is a recursive function whose domain is a type defined as a least fixpoint (an initial algebra), and an unfold is a recursive function whose codomain is defined as a greatest fixpoint (a final coalgebra). However, in lazy languages such as Haskell, least and greatest fixpoints coincide, and are simply called recursive types.

At an abstract level, folds (as well as other structured forms of recursion, such as primitive recursion) enjoy an initiality property among the algebras of the base functor of the domain type. In concrete terms, this makes possible the use of *induction* as a proof technique. Dually,

```
data BTree a = Empty | Node a (BTree a) (BTree a)
type Heap    = BTree
data LTree a = Leaf (Maybe a) | Branch (LTree a) (LTree a)


unfoldBTree :: (b -> (Either (a,b,b) ())) -> b -> BTree a
unfoldBTree g x = case (g x) of
            Right ()     -> Empty
            Left (y,l,r) -> Node y (unfoldBTree g l) (unfoldBTree g r)

foldBTree :: (a -> b -> b -> b) -> b -> BTree a -> b
foldBTree f e Empty        = e
foldBTree f e (Node x l r) = f x (foldBTree f e l) (foldBTree f e r)


unfoldLTree :: (b -> (Either (b,b) (Maybe a))) -> b -> LTree a
unfoldLTree g x = case (g x) of
            Right y    -> Leaf y
            Left (l,r) -> Branch (unfoldLTree g l) (unfoldLTree g r)

foldLTree :: (b->b->b) -> ((Maybe a)->b) -> LTree a -> b
foldLTree f e (Leaf x)     = e x
foldLTree f e (Branch l r) = f (foldLTree f e l) (foldLTree f e r)
```

Table 1. Types and recursion patterns for binary trees

unfolds are final coalgebras; techniques for reasoning about unfolds include *fixpoint induction* and *coinduction* [5].

Unicity. The *program calculation* approach is based on the use of initiality and finality directly as an equational proof principle. Both properties can be formulated in the same framework, as *universal* or *unicity* properties. In this paper we generally adhere to the equational style for proofs, but often resort to induction for the sake of simplicity (in particular when none of the sides of the equality one wants to prove is directly expressed using a recursion pattern, applying a unicity property may require substantial manipulation of the expressions). See [4] for a study of program calculation carried out purely by using fusion.

We give below the unicity properties that we shall require in the rest of the paper, for the foldr, unfoldLTree, and unfoldBTree operators. A weaker *fusion* law for foldr is also shown, which is easily derived from unicity.

$$
\begin{array}{ll}
 & f = \mathsf{foldr}\ g\ e \\
\Leftrightarrow & \quad \{\text{unicity-foldr}\} \\
 & \begin{cases} f\ [\,] = e \\ \text{for all } x, xs, \\ \quad f\ (x : xs) = g\ x\ (f\ xs) \end{cases}
\end{array}
\qquad\qquad
\begin{array}{ll}
 & h \circ \mathsf{foldr}\ g\ e = \mathsf{foldr}\ g'\ e' \\
\Leftarrow & \quad \{\text{foldr-fusion}\} \\
 & \begin{cases} h\ e = e' \\ h\ \circ (g\ x) = (g'\ x) \circ h \end{cases}
\end{array}
$$

$$
\begin{array}{ll}
 & f = \mathsf{unfoldLTree}\ g \\
\Leftrightarrow & \quad \{\text{unicity-unfoldLTree}\} \\
 & \text{for all } x, \\
 & f\ x = \mathsf{case}\ (g\ x)\ \mathsf{of} \\
 & \quad \mathsf{Right}\ y \rightarrow \mathsf{Leaf}\ y \\
 & \quad \mathsf{Left}\ (l, r) \rightarrow \mathsf{Branch}\ (f\ l)\ (f\ r)
\end{array}
\qquad
\begin{array}{ll}
 & f = \mathsf{unfoldBTree}\ g \\
\Leftrightarrow & \quad \{\text{unicity-unfoldBTree}\} \\
 & \text{for all } x, \\
 & f\ x = \mathsf{case}\ (g\ x)\ \mathsf{of} \\
 & \quad \mathsf{Right}\ () \rightarrow \mathsf{Empty} \\
 & \quad \mathsf{Left}\ (y, l, r) \rightarrow \mathsf{Node}\ y\ (f\ l)\ (f\ r)
\end{array}
$$

We direct the reader to [7] for an extensive introduction to the field of program calculation.

Hylomorphisms. The composition of a fold over a regular type T with an unfold of that type is a recursive function whose recursion tree is shaped in the same way as T. Such a definition can be *deforested* [11], i.e. the construction of the intermediate data-structures can be eliminated, yielding a direct recursive definition. As an example, the definition $h = (\mathsf{foldLTree}\ f\ e) \circ (\mathsf{unfoldLTree}\ g)$ can be deforested to give:

```
h x = case (g x) of
      Right y    -> e y
      Left (l,r) -> f (foldLTree f e (unfoldLTree g l))
                      (foldLTree f e (unfoldLTree g r))
```

or

```
h x = case (g x) of
      Right y    -> e y
      Left (l,r) -> f (h l) (h r)
```

This corresponds to a new generic recursion pattern, called a *hylomorphism*. Hylomorphisms do not possess a unicity property, but they are still useful for reasoning about programs, using the properties of their fold and unfold components.

In particular, hylomorphisms are useful for capturing the structure of functions that are not directly defined by structured recursion or co-recursion, as is the case of the divide-and-conquer sorting algorithms: the unfold component takes the unsorted list and constructs a tree; the fold iterates over this structure to produce the sorted list.

In the reverse direction, transforming an arbitrary recursive definition into a hylomorphism [8] is a *forestation* process, which makes explicit the construction of intermediate data structures (this is useful for *program understanding*).

The sorting algorithms introduced in the previous section were studied as hylomorhpisms in [1]. In the present paper we use this hylomorphic structure to calculate these algorithms from a common specification.

4 Sorting by Insertion

In the rest of the paper we will repeatedly apply the following principles. Consider a type constructor C and the following functions:

$$
\begin{aligned}
\mathsf{istC} &: a \rightarrow \mathrm{C}\ a \rightarrow \mathrm{C}\ a \\
\mathsf{C2list} &: \mathrm{C}\ a \rightarrow [a]
\end{aligned}
$$

The idea is that C a is a container type for elements of type a (typically a tree-shaped type); istC inserts an element in a container to give a new container; and $\mathsf{C2list}$ converts a container into a sorted list of type a.

A generic sorting algorithm can then be defined, with a container acting as intermediate data-structure. The idea is that elements are inserted one by one by folding over the list; a sorted list is then obtained using $\mathsf{C2list}$. $\varepsilon :: \mathrm{C}\ a$ is an appropriate "empty value".

$$\mathsf{isortC} = \mathsf{C2list} \circ (\mathsf{foldr}\ \mathsf{istC}\ \varepsilon)$$

It is easy to see that the algorithm is correct if the intermediate data-structure contains exactly the same elements as the initial list, and $\mathsf{C2list}$ somehow produces a sorted list from the elements in the intermediate structure. This can be formalized by constructing a proof of equivalence to insertion sort, which gives necessary conditions for the algorithm to be correct.

$$\begin{array}{ll} & \mathsf{C2list} \circ (\mathsf{foldr}\ \mathsf{istC}\ \varepsilon) = \mathsf{foldr}\ \mathsf{insert}\ [\,] \\ \Leftrightarrow & \quad \{\text{unicity-foldr}\} \\ & \begin{cases} \mathsf{C2list}\ (\mathsf{foldr}\ \mathsf{istC}\ \varepsilon\ [\,]) = [\,] \\ \mathsf{C2list}\ (\mathsf{foldr}\ \mathsf{istC}\ \varepsilon\ (x : xs)) = \mathsf{insert}\ x\ (\mathsf{C2list}\ (\mathsf{foldr}\ \mathsf{istC}\ \varepsilon\ xs)) \end{cases} \\ \Leftrightarrow & \quad \{\text{def. foldr}\} \\ & \begin{cases} \mathsf{C2list}\ \varepsilon = [\,] \\ \mathsf{C2list}\ (\mathsf{istC}\ x\ (\mathsf{foldr}\ \mathsf{istC}\ \varepsilon\ xs)) = \mathsf{insert}\ x\ (\mathsf{C2list}\ (\mathsf{foldr}\ \mathsf{istC}\ \varepsilon\ xs)) \end{cases} \end{array}$$

Alternatively one can use fusion, which leads to stronger conditions:

$$\begin{array}{ll} & \mathsf{C2list} \circ (\mathsf{foldr}\ \mathsf{istC}\ \varepsilon) = \mathsf{foldr}\ \mathsf{insert}\ [\,] \\ \Leftarrow & \quad \{\text{foldr fusion}\} \\ & \begin{cases} \mathsf{C2list}\ \varepsilon = [\,] \\ \mathsf{C2list}\ \circ (\mathsf{istC}\ x) = (\mathsf{insert}\ x) \circ \mathsf{C2list} \end{cases} \end{array}$$

Thus for each concrete container type it is sufficient to prove equation 2 and one of 3 or 4 to establish that the corresponding function isortC is indeed a sorting algorithm:

$$\mathsf{C2list}\ \varepsilon = [\,] \tag{2}$$

$$\mathsf{C2list} \circ (\mathsf{istC}\ x) = (\mathsf{insert}\ x) \circ \mathsf{C2list} \tag{3}$$

$$\mathsf{C2list}\ (\mathsf{istC}\ x\ (\mathsf{foldr}\ \mathsf{istC}\ \varepsilon\ xs)) = \mathsf{insert}\ x\ (\mathsf{C2list}\ (\mathsf{foldr}\ \mathsf{istC}\ \varepsilon\ xs)) \tag{4}$$

Note that together, equations 2 and 3 mean that $\mathsf{C2list}$ is a homomorphism between the structures $(C\ a, \mathsf{istC}, \varepsilon)$ and $([a], \mathsf{insert}, [\,])$.

It is well-known that the introduction of accumulators to produce tail-recursive functions [2] explores the properties of an underlying monoid. In the present case this allows for the intermediate structures of our sorting algorithm to be constructed in a rightwards fashion (even if the function $\mathsf{foldr}\ \mathsf{ist}\ \varepsilon$ is not a list homomorphism). A tail-recursive version of isortC can be derived by a standard transformation based on fusion.

We start by writing a specification for this function isortC_t.

$$\begin{array}{c} \mathsf{isortC}_t\ :\ [a] \to \mathrm{C}\ a \to [a] \\ \mathsf{isortC}_t\ l\ y = (\mathsf{isort}\ l) \odot (\mathsf{C2list}\ y) \end{array}$$

The tail-recursive function uses an extra accumulator argument of the chosen container type. In the call $\mathsf{isortC}_t\ l\ y$, l is the list that remains to be sorted, and the accumulator y

contains elements already inserted in the container. The right-hand side of the equality states how the final result can be obtained using insertion sort and the conversion of y to a list.

The following definition

$$\begin{aligned}\mathsf{isortC}_t = \mathsf{foldr}\ \mathsf{istC}'\ \mathsf{C2list}\\ \quad \mathsf{where}\ \mathsf{istC}'\ x\ f\ y = f\ (\mathsf{istC}\ x\ y)\end{aligned}$$

satisfies the specification (proof is given in Appendix A.1). Then

$$\mathsf{isortC}_t\ l\ \varepsilon = \mathsf{iSort}\ l \tag{5}$$

which is an immediate consequence of the specification and eq. (2) above.

This higher-order fold can be written with explicit recursion as

```
isortC_t :: (Ord a) => [a] -> C a -> [a]
isortC_t [] y     = C2list y
isortC_t (x:xs) y = isortC_t xs (istC x y)
```

An alternative version of this can be defined, which separates the tail-recursive construction of the intermediate structure from its conversion to a sorted list:

$$\mathsf{isortC}'\ l = \mathsf{C2list}\ (\mathsf{foldr}\ \mathsf{istC}'\ \mathsf{id}\ l\ \varepsilon)$$

The following equivalence is straightforward to establish, which proves that isortC' is also a sorting algorithm:

$$\mathsf{isortC}'\ l = \mathsf{iSortC}_t\ l\ \varepsilon \tag{6}$$

We now outline the approach followed in the next sections of the paper. The container type and its empty value, together with the functions istC and C2list, will be instantiated to produce three different insertion-based algorithms (which will be proved correct by calculating eqs. 2, and 3 or 4 above). C2list will always have the same generic definition: it is a fold that merges ($\odot$) together the wrapped contents of each node with the recursive results on the subtrees. This may then be refined, which is justified by a structural invariant satisfied by the trees constructed by f.

These programs are of the form $\mathsf{C2list} \circ f$ where f is a *fold* over lists. f will finally be transformed into an *unfold* of the tree type. The correctness of this step is established by appealing to both initiality and finality arguments. Each algorithm thus obtained is a *hylomorphism* that can be deforested to produce a well-known sorting algorithm.

5 A Derivation of Merge Sort

Our first concrete sorting algorithm based on insertion into an intermediate structure uses *leaf-labelled* binary trees. This is given in Table 2. We remark that to cover the case of the empty list, a Maybe type is used in the leaves of the trees.

Proposition 1. isortLT *is a sorting algorithm.*

```
isortLT  = LT2list ∘ buildLT
buildLT = foldr istLT (Leaf Nothing)

LT2list = foldLTree (⊙) t
            where t Nothing = [ ]
                  t (Just x) = [x]

istLT x (Leaf Nothing) = Leaf (Just x)
istLT x (Leaf(Just y))  = Branch (Leaf (Just x)) (Leaf(Just y))
istLT x (Branch l r)    = Branch (istLT x r) l
```

Table 2. Sorting by insertion in a leaf tree

Proof. We instantiate eqs. (2) and (3). Note that the empty value here is $\varepsilon = \mathsf{Leaf\ Nothing}$.

$$\begin{aligned}\mathsf{LT2list\ (Leaf\ Nothing)} &= [\,]\\ \mathsf{LT2list} \circ (\mathsf{istLT}\ x) &= (\mathsf{insert}\ x) \circ \mathsf{LT2list}\end{aligned}$$

The first equality if true by definition; the second can be proved by induction, or alternatively using fusion. The latter proof is given in Appendix A.2. Together these equations establish that LT2list is a homomorphism between the structures ($\mathsf{LTree}\ a, \mathsf{istLT}, \mathsf{Leaf\ Nothing}$) and ($[a], \mathsf{insert}, [\,]$).

It is also easy to see that the intermediate tree is *balanced*: the difference between the heights of the subtrees of a node is never greater than one, since subtrees are swapped at each insertion step. Note that the insertion function istLT was carefully designed with efficiency in mind, which grants execution in time $O(N \lg N)$; other solutions would still lead to sorting algorithms, albeit less efficient.

Proposition 2. *The trees constructed by* buildLT *are balanced.*

Proof. It can be proved by induction on the structure of the argument list that either the subtrees of the constructed tree have the same height, or the height of the left subtree is greater than the height of the right subtree by one unit. The function istLT preserves this invariant.

The next transformation step applies to the function that constructs the intermediate tree. An alternative way of constructing a balanced tree is by *unfolding*: the initial list is traversed and its elements placed alternately in two subsequences, which are then used as arguments to recursively construct the subtrees. Note that the sequences will have approximately the same length. For singular and empty lists, leaves are returned.

```
unfoldmsort = unfoldLTree g
              where g [ ] = Right Nothing
                    g [x] = Right (Just x)
                    g xs = Left (maux xs)
                    maux [ ] = ([ ], [ ])
                    maux (h : t) = (h : b, a) where (a, b) = maux t
```

```
isortH  = H2list ∘ buildH
buildH = foldr istH Empty

H2list  = foldr aux []
          where aux x l r = x : (l ⊙ r)

istH x Empty        = Node x Empty Empty
istH x (Node y l r) | x < y    =    Node x (istH y r) l
                    | otherwise =    Node y (istBST x r) l
```

Table 3. Sorting by insertion in a heap

Proposition 3. *The above function constructs the same intermediate trees as those obtained by folding over the argument list:*

$$\mathsf{foldr\ istLT\ (Leaf\ Nothing)} = \mathsf{unfoldmsort}$$

Proof. We use the unicity property of leaf-tree unfolds:

$$\begin{array}{ll}
& \mathsf{foldr\ istLT\ (Leaf\ Nothing)} = \mathsf{unfoldLTree}\ g \\
\Leftrightarrow & \quad \{\text{unicity-unfoldLTree}\} \\
& \text{for all } x, \\
& (\mathsf{foldr\ istLT\ (Leaf\ Nothing)})\ x = \mathsf{case\ (g}\ x)\ \mathsf{of} \\
& \quad \mathsf{Right}\ y \to \mathsf{Leaf}\ y \\
& \quad \mathsf{Left}\ (l, r) \to \mathsf{Branch\ (f}\ l)\ (\mathsf{f}\ r) \\
\Leftrightarrow & \quad \{\text{by cases}\} \\
& \begin{cases}
\mathsf{Leaf\ Nothing} = \mathsf{Leaf\ Nothing} & \text{if } x = [\,] \\
\mathsf{Leaf(Just}\ h) = \mathsf{Leaf(Just}\ h) & \text{if } x = [h] \\
\mathsf{istLT}\ h_1\ (\mathsf{foldr\ istLT\ (Leaf\ Nothing)}\ (h_2 : t)) & \\
= \mathsf{Branch\ (foldr\ istLT\ (Leaf\ Nothing)}\ l) & \\
\qquad\qquad (\mathsf{foldr\ istLT\ (Leaf\ Nothing)}\ r) & \text{if } x = h_1 : h_2 : t \\
\qquad \text{where } (l, r) = \mathsf{maux}\ (h_1 : h_2 : t) &
\end{cases}
\end{array}$$

And the last equality can be easily proved by induction on the structure of t.

Substituting this in the definition of isortLT yields a hylomorphism that is of course still equivalent to insertion sort. It is immediate to see that this can be deforested, and the result is merge sort:

$$\mathsf{LT2list} \circ \mathsf{unfoldmsort} = \mathsf{msort}$$

6 A Derivation of Heapsort

In the *heapsort* algorithm, one computes the minimum of the list prior to the recursive calls. This will determine that each node of the intermediate structure (the recursion tree) this minimum for some tree; it is thus a binary *node-labelled* tree.

We repeat the program taken for the derivation of the merge sort: we design a function that inserts a single element in the intermediate tree (istH), iterate this function over the initial list (buildH) and then provide a function that recovers the ordered list from the tree (H2list). These functions are shown in Table 3.

Proposition 4. isortH *is a sorting algorithm.*

Proof. We instantiate eqs. (2) and (4). We set $\epsilon = \mathsf{Empty}$, and thus eq. (2) results directly from the definition. For eq. (4), we need to prove that for every list l,

$$\mathsf{insert}\ x\ (\mathsf{H2list}\ (\mathsf{buildH}\ l)) = \mathsf{H2list}\ (\mathsf{istH}\ x\ (\mathsf{buildH}\ l)).$$

In order to prove this, we rely on the fact that trees generated by buildH are always *heaps*, i.e. the root element is the least of the tree. The complete derivation is presented in appendix B (Propositions 8 and 1).

Note that in order to prove the correctness of this algorithm, we cannot rely on the strongest hypothesis given by eq. 3 (obtained from the use of the fusion law) as we have done for merge sort. The reason for this is that, for an arbitrary tree t,

$$\mathsf{insert}\ x\ (\mathsf{H2list}\ t) \neq \mathsf{H2list}\ (\mathsf{istH}\ x\ t).$$

On the other hand, the weaker requisite given by eq. 4 (obtained by the use of unicity or induction) retains the information that we restrict our attention to trees constructed by buildBST, and these will satisfy the required equality.

We also note that the intermediate tree is again balanced (essentially by the same argument used for merge sort). This means that this sorting algorithm also executes in time $O(N \lg N)$.

It remains to show that the intermediate tree can be constructed coinductively. For that, consider the following function:

$$\begin{aligned}
&\mathsf{unfoldhsort} = \mathsf{unfoldBTree\ g}\\
&\qquad \mathsf{where\ g}\ [\,] \qquad\quad\ = \mathsf{Right}\ ()\\
&\qquad\quad \mathsf{g}\ (x : xs) \quad\ = \mathsf{Left}\ (\mathsf{haux}\ x\ xs)\\
&\qquad\quad \mathsf{haux}\ x\ [\,] \quad\ = (x, [\,], [\,])\\
&\qquad\quad \mathsf{haux}\ x\ (y : ys)\ \mid\ x < m = (x, m : b, a)\\
&\qquad\qquad\qquad\qquad\ \mid\ \mathsf{otherwise} = (m, x : b, a)\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathsf{where}\ (m, a, b) = \mathsf{haux}\ y\ ys
\end{aligned}$$

Proposition 5. *The above function constructs the same intermediate trees as those obtained by folding over the argument list:*

$$\mathsf{buildH} = \mathsf{unfoldhsort}$$

Proof.

$$\begin{aligned}
&\mathsf{unfoldhsort} = \mathsf{buildH}\\
\Leftrightarrow\quad &\{\text{by unicity} - \mathsf{unfoldBTree}\}\\
&\begin{cases}
\mathsf{buildH}\ [\,] = \mathsf{Empty}\\
\mathsf{buildH}\ (x : xs) = \mathsf{Node}\ z\ (\mathsf{buildH}\ a)\ (\mathsf{buildH}\ b)\\
\quad \text{where } (z, a, b) = \mathsf{haux}\ x\ xs
\end{cases}\\
\Leftrightarrow\quad &\{\text{definitions}\}\\
&\begin{cases}
\mathsf{Empty} = \mathsf{Empty}\\
\mathsf{istH}\ x\ (\mathsf{buildH}\ xs) = \mathsf{Node}\ z\ (\mathsf{buildH}\ a)\ (\mathsf{buildH}\ b)\\
\quad \text{where } (z, a, b) = \mathsf{haux}\ x\ xs
\end{cases}
\end{aligned}$$

```
isortBST  = BST2list ∘ buildBST
buildBST = (ap Empty) ∘ bAcc
             where ap x f = f x
                   bAcc = foldr aux id
                   aux x f a = f (istBST x a)

BST2list = foldBTree aux []
             where aux x l r = l ++(x : r)

istBST x (Empty)      = Node x Empty Empty
istBST x (Node y l r) | x < y    =    Node y (istBST x l) r
                      | otherwise  =    Node y l (istBST x r)
```

Table 4. Sorting by insertion in a binary search tree

The second equality is proved by structural induction on xs. For the base case ($xs = [\,]$), it follows directly from evaluating the definitions. For the inductive step ($xs = y : ys$), let us assume that $(z, a, b) = (\mathsf{haux}\ y\ ys)$. The definition of haux tell us that

$$(z', a', b') = (\mathsf{haux}\ x\ (y : ys)) = \begin{cases} (x, z : b, a) & \text{if } x < z, \\ (z, x : b, a) & \text{if } x \geq z. \end{cases}$$

Thus,

$$\begin{array}{ll} & \mathsf{istH}\ x\ (\mathsf{buildH}\ (y : ys)) \\ = & \quad \{\text{def. buildH}\} \\ & \mathsf{istH}\ x\ (\mathsf{istH}\ y\ (\mathsf{buildH}\ ys)) \\ = & \quad \{\text{induction hypotheses}\} \\ & \mathsf{istH}\ x\ (\mathsf{Node}\ z\ (\mathsf{buildH}\ a)\ (\mathsf{buildH}\ b)) \\ = & \quad \{\text{def. istH}\} \\ & \begin{cases} \mathsf{Node}\ x\ (\mathsf{istH}\ z\ (\mathsf{buildH}\ b))\ (\mathsf{buildH}\ a) & \text{if } x < z, \\ \mathsf{Node}\ z\ (\mathsf{istH}\ x\ (\mathsf{buildH}\ b))\ (\mathsf{buildH}\ a) & \text{if } x \geq z. \end{cases} \\ = & \quad \{\text{def. of } z', a', b'\} \\ & \mathsf{Node}\ z'\ (\mathsf{buildH}\ a')\ (\mathsf{buildH}\ b') \end{array}$$

As would be expected, the hylomorphism obtained replacing buildH by unfoldhsort can be deforested, and the result is the original hsort.

$$\mathsf{H2list} \circ \mathsf{unfoldhsort} = \mathsf{hsort}$$

7 A Derivation of Quicksort

In the quicksort algorithm, the activity performed prior to the recursive calls is different from that in heapsort: instead of finding the minimum of the list, the head of the list is used as a *pivot* for splitting the tail. Again, the intermediate structure is a *node-labelled* binary tree. But now, its ordering properties are different — the constructed trees will be *binary search trees*, and it suffices to traverse these trees *in-order* to produce the desired sorted list.

Following the same line as in the derivation of the previous algorithms, we define an algorithm that iteratively inserts elements from a list into a binary tree and then reconstructs the list by the *in-order* traversal. This algorithm is given in Table 4.

Observe that this algorithm iterates on the initial list from left to right (we may think of it as using the Haskell foldl operator, but we write it as a higher-order function using foldr, to exploit the application of the rules presented earlier). This will become evident below when we replace this function by one that constructs the intermediate tree corecursively. For the correctness argument, we know that the order of traversal for the initial list is irrelevant (as shown in Section 4).

Proposition 6. isortBST *is a sorting algorithm.*

Proof. We instantiate eqs. 2 and 4. We set $\epsilon = \mathsf{Empty}$, and thus Equation 2 results directly from the definition. For Equation 4, we need to prove that for every list,

$$\mathsf{insert}\ x\ (\mathsf{BST2list}\ (\mathsf{buildBST}\ l)) = \mathsf{BST2list}\ (\mathsf{istBST}\ x\ (\mathsf{buildBST}\ l)).$$

In order to prove this, we rely on the fact that trees generated by buildBST are always binary search trees. The complete derivation is presented in appendix B (Propositions 10 and 2).

To obtain the well-known quicksort algorithm, we need to replace the iterated insertion function buildBST by an unfold.

$$\begin{aligned}
&\mathsf{unfoldqsort} = \mathsf{unfoldBTree\ g}\\
&\qquad \mathsf{where\ g}\ [\,] &&= \mathsf{Right}\ ()\\
&\qquad\quad \mathsf{g}\ (x : xs) &&= \mathsf{Left}\ (\mathsf{qaux}\ x\ xs)\\
&\qquad\quad \mathsf{qaux}\ x\ [\,] &&= (x, [\,], [\,])\\
&\qquad\quad \mathsf{qaux}\ x\ (y : ys) &&\mid y < x = (x, y : b, a)\\
&\qquad\quad\qquad\qquad &&\mid \mathsf{otherwise} = (m, a, y : b)\\
&\qquad\quad\qquad\qquad\qquad\qquad &&\mathsf{where}\ (a, b) = \mathsf{qaux}\ x\ ys
\end{aligned}$$

Proposition 7. *The above function constructs the same intermediate trees as those obtained by folding over the argument list:*

$$\mathsf{buildBST} = \mathsf{unfoldqsort}$$

Proof.

$$\begin{aligned}
&\mathsf{unfoldqsort} = \mathsf{buildBST}\\
\Leftrightarrow\quad &\{\text{unicity-unfolBTree}\}\\
&\begin{cases}\mathsf{buildBST}\ [\,] = \mathsf{Empty}\\ \mathsf{buildBST}\ (x : xs) = \mathsf{Node}\ x\ (\mathsf{bAcc}\ a\ \mathsf{Empty})\ (\mathsf{bAcc}\ b\ \mathsf{Empty})\\ \quad \mathsf{where}\ (a, b) = \mathsf{qaux}\ x\ xs\end{cases}\\
\Leftrightarrow\quad &\{\text{definitions}\}\\
&\begin{cases}\mathsf{bAcc}\ [\,]\ \mathsf{Empty} = \mathsf{Empty}\\ \mathsf{bAcc}\ (x : xs)\ \mathsf{Empty} = \mathsf{Node}\ x\ (\mathsf{bAcc}\ a\ \mathsf{Empty})\ (\mathsf{bAcc}\ b\ \mathsf{Empty})\\ \quad \mathsf{where}\ (a, b) = \mathsf{qaux}\ x\ xs\end{cases}\\
\Leftrightarrow\quad &\{\text{simplification}\}\\
&\begin{cases}\mathsf{Empty} = \mathsf{Empty}\\ \mathsf{bAcc}\ xs\ (\mathsf{Node}\ x\ \mathsf{Empty}\ \mathsf{Empty}) = \mathsf{Node}\ x\ (\mathsf{bAcc}\ a\ \mathsf{Empty})\ (\mathsf{bAcc}\ b\ \mathsf{Empty})\\ \quad \mathsf{where}\ (a, b) = \mathsf{qaux}\ x\ xs\end{cases}
\end{aligned}$$

We prove the second equality in a slightly strengthened formulation. For every tree $(\mathsf{Node}\ x\ l\ r)$ and list xs,

$$\mathsf{bAcc}\ xs\ (\mathsf{Node}\ x\ l\ r) = \mathsf{Node}\ x\ (\mathsf{bAcc}\ a\ l)\ (\mathsf{bAcc}\ b\ r)$$
$$\text{where } (a, b) = \mathsf{qaux}\ x\ xs$$

By induction on the structure of xs. For the base case ($xs = [\,]$), it follows directly from evaluating the definitions. For the inductive step ($xs = y : ys$), we reason by cases. If $x < y$, then

$$\begin{array}{ll}
 & \mathsf{bAcc}\ (y:ys)\ (\mathsf{Node}\ x\ l'\ r') = \mathsf{Node}\ x\ (\mathsf{bAcc}\ a'\ l')\ (\mathsf{bAcc}\ b'\ r') \\
 & \quad \text{where } (a', b') = \mathsf{qaux}\ x\ (y:ys) \\
\Leftrightarrow & \quad \{\text{definition, } x < y\} \\
 & \mathsf{bAcc}\ ys\ (\mathsf{istBST}\ y\ (\mathsf{Node}\ x\ l'\ r') = \mathsf{Node}\ x\ (\mathsf{bAcc}\ a\ l')\ (\mathsf{bAcc}\ (y:b)\ r') \\
 & \quad \text{where } (a, b) = \mathsf{qaux}\ x\ ys \\
\Leftrightarrow & \quad \{\text{definition, } x < y\} \\
 & \mathsf{bAcc}\ ys\ (\mathsf{Node}\ x\ l'\ (\mathsf{istBST}\ y\ r') = \mathsf{Node}\ x\ (\mathsf{bAcc}\ a\ l')\ (\mathsf{bAcc}\ b\ (\mathsf{istBST}\ y\ r')) \\
 & \quad \text{where } (a, b) = \mathsf{qaux}\ x\ ys \\
\Leftrightarrow & \quad \{\text{induction hypoptheses}\} \\
 & \mathsf{bAcc}\ ys\ (\mathsf{Node}\ x\ l'\ (\mathsf{istBST}\ y\ r') = \mathsf{bAcc}\ ys\ (\mathsf{Node}\ x\ l'\ (\mathsf{istBST}\ y\ r')) \\
 & \quad \text{where } (a, b) = \mathsf{qaux}\ x\ ys
\end{array}$$

Similarly for the case ($x \geq y$). This concludes the proof.

We conclude with the statement that the hylomorphism obtained is, as intended, the forested version of the original quicksort algorithm.

$$\mathsf{BST2list} \circ \mathsf{unfoldqsort} = \mathsf{qsort}$$

8 Conclusion

To sum up our approach, we take the list homomorphism ($\mathsf{foldr}\ \mathsf{insert}\ [\,]$) as the specification of "sorting". The efficiency of computing it directly leftwards or rightwards can be improved by using an intermediate tree structure (constructed either by folding or unfolding). The tree can in general be converted to the result of the homomorphism by folding with the $\odot$ operator, and this fold may be optimized depending on properties of the intermediate structures.

This first step yields a sorting algorithm based on insertion by folding; the second step consists in the transformation of the fold that construct the intermediate structure into an unfold. The correctness of this step is established appealing to both initiality and finality arguments.

Apart from the proofs of correctness which as far as we know are new, the contributions of this paper include (two versions of) a *generic* sorting algorithm, of which 3 concretizations are used.

This study illustrates the strengths of the "program calculation" style of reasoning, in particular the simplicity of using the unicity property of unfolds as an alternative to using coinductive principles based on bissimulations, and more generally the structural aspects of proofs. We firmly believe however that inductive proofs are much simpler to carry out in many situations.

As a concluding remark, we would like to emphasize the role played by structural invariants in this case study. We could say that if they are not crucial to some calculations, invariants provide a much more natural setting for conducting them. Moreover, this study opens the way to a richer interplay between the invariants and recursion patterns – a topic that is not explored in this paper, but is being currently investigated by the authors.

References

1. Lex Augusteijn. Sorting morphisms. In S. Swierstra, P. Henriques, and J. Oliveira, editors, *Advanced Functional Programming*, LNCS Tutorials, pages 1–27. Springer-Verlag, 1998.
2. Richard Bird. The promotion and accumulation strategies in transformational programming. *ACM Trans. Program. Lang. Syst.*, 6(4):487–504, 1984.
3. Richard Bird. An Introduction to the Theory of Lists. In M. Broy, editor, *Logic of Programming and Calculi of Descrete Design*. Springer-Verlag, 1987.
4. Alcino Cunha and Jorge Sousa Pinto. Point-free program transformation. *Fundamenta Informaticae*, 66(4), April-May 2005. Special Issue on Program Transformation.
5. J. Gibbons and G. Hutton. Proof Methods for Structured Corecursive Programs. In *Proceedings of the 1st Scottish Functional Programming Workshop*, 1999.
6. Jeremy Gibbons. The Third Homomorphism Theorem. *Journal of Functional Programming*, 1995. Functional Pearl.
7. Jeremy Gibbons. Calculating Functional Programs. In *Proceedings of ISRG/SERG Research Colloquium*. School of Computing and Mathematical Sciences, Oxford Brookes University, 1997.
8. Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Deriving structural hylomorphisms from recursive definitions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pages 73–82. ACM Press, 1996.
9. Lambert Meertens. Paramorphisms. Technical Report RUU-CS-90-4, Utrecht University, Department of Computer Science, 1990.
10. Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'91)*, volume 523 of *LNCS*. Springer-Verlag, 1991.
11. P. Wadler. Deforestation: Transforming programs to eliminate trees. In *ESOP '88: the second European Symposium on Programming*, pages 344–358, 1988.

A Proofs and Calculations

A.1

The function

$$\begin{aligned}\mathsf{isortC}_t &= \mathsf{foldr}\ \mathsf{istC}'\ \mathsf{C2list}\\ &\quad \text{where } \mathsf{istC}'\ x\ f\ y = f\ (\mathsf{istC}\ x\ y)\end{aligned}$$

satisfies the specification

$$\begin{aligned}\mathsf{isortC}_t\ &:\ [a] \to \mathrm{C}\ a \to [a]\\ \mathsf{isortC}_t\ l\ y &= (\mathsf{isort}\ l) \odot (\mathsf{C2list}\ y)\end{aligned}$$

Proof. The specification can be rewritten as

$$\mathsf{isortC}_t\ l\ y = (\mathsf{isort}\ l) \oplus y$$

or

$$\mathsf{isortC}_t = (\oplus) \circ \mathsf{isort}$$

with the $\oplus$ operator defined as

$$s \oplus y = s \odot (\mathsf{C2list}\ y)$$

This appeals to the use of the fusion law since isort is defined as a fold.

$$\begin{array}{ll}
 & \mathsf{isortC}_t = (\oplus) \circ \mathsf{isort}\\
\Leftrightarrow & \quad \{\text{definitions}\}\\
 & \mathsf{foldr}\ \mathsf{istC}'\ \mathsf{C2list} = (\oplus) \circ (\mathsf{foldr}\ \mathsf{insert}\ [\,])\\
\Leftarrow & \quad \{\mathsf{foldr}\text{ fusion, with }\oplus\text{ strict}\}\\
 & \begin{cases} (\oplus)\ [\,] = \mathsf{C2list} \\ (\oplus) \circ (\mathsf{insert}\ x) = (\mathsf{istC}'\ x) \circ (\oplus) \end{cases}\\
\Leftrightarrow & \quad \{\eta\text{-expansion}\}\\
 & \begin{cases} [\,] \oplus y = \mathsf{C2list}\ y \\ (\mathsf{insert}\ x\ l) \oplus y = \mathsf{istC}'\ x\ ((\oplus)\ l)\ y \end{cases}\\
\Leftrightarrow & \quad \{\text{def. }\oplus\text{, properties of }\odot\text{, def. istC'}\}\\
 & \begin{cases} \mathsf{C2list}\ y = \mathsf{C2list}\ y \\ (\mathsf{insert}\ x\ l) \oplus y = l \oplus (\mathsf{istC}\ x\ y) \end{cases}\\
\Leftrightarrow & \quad \{\text{eq.(1), def. }\oplus\}\\
 & [x] \odot l \odot (\mathsf{C2list}\ y) = l \odot (\mathsf{C2list}\ (\mathsf{istC}\ x\ y))\\
\Leftrightarrow & \quad \{\text{eq. (3)}\}\\
 & [x] \odot l \odot (\mathsf{C2list}\ y) = l \odot (\mathsf{insert}\ x\ (\mathsf{C2list}\ y))\\
\Leftrightarrow & \quad \{\text{eq.(1), properties of }\odot\}\\
 & [x] \odot l \odot (\mathsf{C2list}\ y) = [x] \odot l \odot (\mathsf{C2list}\ y)
\end{array}$$

A.2

We prove

$$\mathsf{LT2list} \circ (\mathsf{istLT}\ x) = (\mathsf{insert}\ x) \circ \mathsf{LT2list}$$

first by calculation, and then using induction.

```
paraLTree :: ((LTree a)->b->(LTree a)->b->b)-> ((Maybe a)->b)-> LTree a-> b
paraLTree f g (Leaf x)     = g x
paraLTree f g (Branch l r) = f l (paraLTree f g l) r (paraLTree f g r)
```

$$
\begin{array}{l|l}
\quad h = \mathsf{paraLTree}\ f\ g & \quad h \circ (\mathsf{paraLTree}\ f\ g) = \mathsf{paraLTree}\ a\ b \\
\Leftrightarrow \quad \{\text{unicity-paraLTree}\} & \Leftrightarrow \quad \{\text{paraLTree-fusion}\} \\
\quad \begin{cases} h \circ \mathsf{Leaf} = g \\ \text{for all } l, r, \\ \quad h\ (\mathsf{Branch}\ l\ r) = f\ l\ (h\ l)\ r\ (h\ r) \end{cases} & \quad h \text{ strict } \wedge\ h \circ g = b \wedge \\
 & \quad h(f\ l\ l'\ r\ r') = a\ l\ (h\ l')\ r\ (h\ r')
\end{array}
$$

Table 5. The list paramorphism recursion pattern and laws

Proof by Calculation. It is easy to see that the insertion function istLT cannot be written as a fold over trees, since it uses one of the subtrees unchanged (insertion will proceed recursively in the other subtree). This is a typical example of a situation where iteration is not sufficient: primitive recursion is required. This has been studied as the *paramorphism* recursion pattern [9]. The operator in Table 5 embodies this pattern for leaf-trees. The corresponding unicity property and fusion law [10] are also shown in the table.

The function istLT x can now be written as the following paramorphism of leaf trees

$$
\begin{array}{l}
\mathsf{istLT}\ x = \mathsf{paraLTree}\ f\ g \\
\qquad \text{where}\, g\ \mathsf{Nothing} = \mathsf{Leaf}(\mathsf{Just}\ x) \\
\qquad\qquad g\ (\mathsf{Just}\ y) = \mathsf{Branch}\ (\mathsf{Leaf}(\mathsf{Just}\ x))\ (\mathsf{Leaf}(\mathsf{Just}\ y)) \\
\qquad\qquad f\ l\ l'\ r\ r' \ = \mathsf{Branch}\ r'\ l
\end{array}
$$

We use the following strategy: we apply fusion to prove the left-hand side of the equality equivalent to a new paramorphism; subsequently we prove by unicity that the right-hand side of the equality is also equivalent to this paramorphism.

$$
\begin{array}{ll}
 & \mathsf{LT2ist} \circ (\mathsf{istLT}\ x) = \mathsf{paraLTree}\ a\ b \\
\Leftrightarrow & \quad \{\text{def. of istLT } x \text{ as a paramorphism}\} \\
 & \mathsf{LT2list} \circ (\mathsf{paraLTree}\ f\ g) = \mathsf{paraLTree}\ a\ b \\
\Leftarrow & \quad \{\text{paraLTree-fusion, with LT2list strict}\} \\
 & \begin{cases} \mathsf{LT2list} \circ g = b \\ \mathsf{LT2list}(f\ l\ l'\ r\ r') = a\ l\ (\mathsf{LT2list}\ l')\ r\ (\mathsf{LT2list}\ r') \end{cases} \\
\Leftrightarrow & \quad \{\eta\text{-expansion, def. } f, g\} \\
 & \begin{cases} \mathsf{LT2list}\ (\mathsf{Leaf}(\mathsf{Just}\ x)) = b\ \mathsf{Nothing} \\ \mathsf{LT2list}\ (\mathsf{Branch}\ (\mathsf{Leaf}(\mathsf{Just}\ x))\ (\mathsf{Leaf}(\mathsf{Just}\ y))) = b\ (\mathsf{Just}\ y) \\ \mathsf{LT2list}\ (\mathsf{Branch}\ r'\ l) = a\ l\ (\mathsf{LT2list}\ l')\ r\ (\mathsf{LT2list}\ r') \end{cases} \\
\Leftrightarrow & \quad \{\text{def. LT2list}\} \\
 & \begin{cases} [x] = b\ \mathsf{Nothing} \\ [x] \odot [y] = b\ (\mathsf{Just}\ y) \\ (\mathsf{LT2list}\ r') \odot (\mathsf{LT2list}\ l) = a\ l\ (\mathsf{LT2list}\ l')\ r\ (\mathsf{LT2list}\ r') \end{cases}
\end{array}
$$

We are thus led to define

$$
\begin{array}{l}
b\ \mathsf{Nothing} = [x] \\
b\ (\mathsf{Just}\ y) = [x] \odot [y] \\
a\ l\ l''\ r\ r'' = r'' \odot (\mathsf{LT2list}\ l)
\end{array}
$$

It remains to prove $(\mathsf{insert}\ x) \circ \mathsf{LT2list} = \mathsf{paraLTree}\ a\ b$. Again we proceed by using fusion; the trick is now to write the fold LT2list as a paramorphism (this is always possible since it is a particular case).

$$
\begin{array}{ll}
 & (\mathsf{insert}\ x) \circ \mathsf{LT2list} = \mathsf{paraLTree}\ a\ b \\
\Leftrightarrow & \quad \{\text{unicity-paraLTree, with } (\mathsf{insert}\ x) \text{ strict}\} \\
 & \begin{cases} (\mathsf{insert}\ x) \circ \mathsf{LT2list} \circ \mathsf{Leaf} = b \\ (\mathsf{insert}\ x)\ (\mathsf{LT2list}\ (\mathsf{Branch}\ l\ r)) = a\ l\ (\mathsf{insert}\ x\ (\mathsf{LT2list}\ l))\ r\ (\mathsf{insert}\ x\ (\mathsf{LT2list}\ r)) \end{cases} \\
\Leftrightarrow & \quad \{\text{def. of } \mathsf{LT2list}\} \\
 & \begin{cases} (\mathsf{insert}\ x) \circ g = b \\ \mathsf{insert}\ x\ (f\ l\ (\mathsf{LT2list}\ l)\ r\ (\mathsf{LT2list}\ r)) = a\ l\ (\mathsf{insert}\ x\ (\mathsf{LT2list}\ l))\ r\ (\mathsf{insert}\ x\ (\mathsf{LT2list}\ r)) \end{cases} \\
 & \text{where} \\
 & \quad g\ \mathsf{Nothing} = [\,] \\
 & \quad g\ (\mathsf{Just}\ y) = [y] \\
 & \quad f\ l\ l'\ r\ r' = l' \odot r' \\
\Leftrightarrow & \quad \{\eta\text{-expansion, def. of } f, g, a, b\} \\
 & \begin{cases} \mathsf{insert}\ x\ [\,] = [x] \\ \mathsf{insert}\ x\ [y] = [x] \odot [y] \\ \mathsf{insert}\ x\ (\mathsf{LT2list}\ l \odot \mathsf{LT2list}\ r) = (\mathsf{insert}\ x\ (\mathsf{LT2list}\ r)) \odot (\mathsf{LT2list}\ l) \end{cases} \\
\Leftrightarrow & \quad \{(1) \text{ and properties of } \odot\} \\
 & \begin{cases} [x] = [x] \\ [x] \odot [y] = [x] \odot [y] \\ [x] \odot (\mathsf{LT2list}\ l) \odot (\mathsf{LT2list}\ r) = [x] \odot (\mathsf{LT2list}\ l) \odot (\mathsf{LT2list}\ r) \end{cases}
\end{array}
$$

Proof by Induction.

1. $c = \mathsf{Leaf\ Nothing}$

$$
\begin{array}{ll}
 & \mathsf{LT2list}\ (\mathsf{istLT}\ x\ (\mathsf{Leaf\ Nothing})) = \mathsf{insert}\ x\ (\mathsf{LT2list}\ (\mathsf{Leaf\ Nothing})) \\
\Leftrightarrow & \quad \{\text{def. istLT, LT2list}\} \\
 & \mathsf{LT2list}\ (\mathsf{Leaf}\ (\mathsf{Just}\ x)) = \mathsf{insert}\ x\ [\,] \\
\Leftrightarrow & \quad \{\text{def. LT2list, insert}\} \\
 & [x] = [x]
\end{array}
$$

2. $c = \mathsf{Leaf}\ (\mathsf{Just}\ y)$

$$
\begin{array}{ll}
 & \mathsf{LT2list}\ (\mathsf{istLT}\ x\ (\mathsf{Leaf}\ (\mathsf{Just}\ y))) = \mathsf{insert}\ x\ (\mathsf{LT2list}\ (\mathsf{Leaf}\ (\mathsf{Just}\ y))) \\
\Leftrightarrow & \quad \{\text{def. istLT, LT2list}\} \\
 & \mathsf{LT2list}\ (\mathsf{Branch}\ (\mathsf{Leaf}\ (\mathsf{Just}\ x))\ (\mathsf{Leaf}\ (\mathsf{Just}\ y))) = \mathsf{insert}\ x\ [y] \\
\Leftrightarrow & \quad \{\text{def. LT2list, Spec. theorem}\} \\
 & (\mathsf{LT2list}\ (\mathsf{Leaf}\ (\mathsf{Just}\ x))) \odot (\mathsf{LT2list}\ (\mathsf{Leaf}\ (\mathsf{Just}\ y))) = (\mathsf{wrap}\ x) \odot [y] \\
\Leftrightarrow & \quad \{\text{def. LT2list, wrap}\} \\
 & [x] \odot [y] = [x] \odot [y]
\end{array}
$$

3. $c = \mathsf{Branch}\ l\ r$

$$
\begin{aligned}
& \mathsf{LT2list}\ (\mathsf{istLT}\ x\ (\mathsf{Branch}\ l\ r)) = \mathsf{insert}\ x\ (\mathsf{LT2list}\ (\mathsf{Branch}\ l\ r)) \\
\Leftrightarrow \quad & \{\text{def. istLT, LT2list}\} \\
& \mathsf{LT2list}\ (\mathsf{Branch}\ (\mathsf{istLT}\ x\ r)\ l) = \mathsf{insert}\ x\ ((\mathsf{LT2list}\ l) \odot (\mathsf{LT2list}\ r)) \\
\Leftrightarrow \quad & \{\text{def. LT2list, Spec. theorem}\} \\
& (\mathsf{LT2list}\ (\mathsf{istLT}\ x\ r)) \odot (\mathsf{LT2list}\ l) = (\mathsf{wrap}\ x) \odot ((\mathsf{LT2list}\ l) \odot (\mathsf{LT2list}\ r)) \\
\Leftrightarrow \quad & \{\text{induction, commut. } \odot\} \\
& (\mathsf{insert}\ x\ (\mathsf{LT2list}\ r)) \odot (\mathsf{LT2list}\ l) = (\mathsf{wrap}\ x) \odot ((\mathsf{LT2list}\ r) \odot (\mathsf{LT2list}\ l)) \\
\Leftrightarrow \quad & \{\text{Spec. theorem, assoc. } \odot\} \\
& (\mathsf{wrap}\ x) \odot (\mathsf{LT2list}\ r) \odot (\mathsf{LT2list}\ l) = (\mathsf{wrap}\ x) \odot (\mathsf{LT2list}\ r) \odot (\mathsf{LT2list}\ l)
\end{aligned}
$$

B Tree Invariants

In order to prove certain equalities, it is convenient to introduce a notion of *invariant* that captures properties satisfied by the intermediate structures. These invariants are defined structurally on the data types.

For every predicate $p : A \to \mathsf{Bool}$, we consider the following inductive predicates:

$$(\mathsf{AllL}\ p\ [\,])$$
$$\forall x, xs.(p\ x) \wedge (\mathsf{AllL}\ p\ xs) \Rightarrow (\mathsf{AllL}\ p\ (x : xs))$$

$$(\mathsf{AllT}\ p\ \mathsf{Empty})$$
$$\forall x, l, r.(p\ x) \wedge (\mathsf{AllT}\ p\ l) \wedge (\mathsf{AllT}\ p\ r) \Rightarrow (\mathsf{AllT}\ p\ (\mathsf{Node}\ x\ l\ r)$$

$$(\mathsf{BST}\ \mathsf{Empty})$$
$$\forall x, l, r.(\mathsf{AllT}\ (< x)\ l) \wedge (\mathsf{AllT}\ (\geq x)\ r) \wedge (\mathsf{BST}\ l) \wedge (\mathsf{BST}\ r) \Rightarrow (\mathsf{BST}\ (\mathsf{Node}\ x\ l\ r))$$

$$(\mathsf{HEAP}\ \mathsf{Empty})$$
$$\forall x, l, r.(\mathsf{AllT}\ (\geq x)\ l) \wedge (\mathsf{AllT}\ (\geq x)\ r) \wedge (\mathsf{HEAP}\ l) \wedge (\mathsf{HEAP}\ r) \Rightarrow (\mathsf{HEAP}\ (\mathsf{Node}\ x\ l\ r))$$

Let us start stating some simple properties concerning lists and trees.

Lemma 1. *For every values x, y and lists $l_1.l_2$, we have:*

1. $x < y \Rightarrow \mathsf{insert}\ x\ (l_1 +\!\!+ [y] +\!\!+ l_2) = (\mathsf{insert}\ x\ l_1) +\!\!+ [y] +\!\!+ l_2$
2. $(\mathsf{AllL}\ (\leq\ x)\ l_1) \Rightarrow \mathsf{insert}\ x\ (l_1 +\!\!+ l_2) = l_1 +\!\!+ (\mathsf{insert}\ x\ l_2)$
3. $(\forall x.p\ x \Rightarrow q\ x) \Rightarrow \mathsf{AllL}\ p\ l_1 \Rightarrow \mathsf{AllL}\ q\ l_1$
4. $(\mathsf{AllL}\ p\ (l_1 +\!\!+ l_2)) \Leftrightarrow (\mathsf{AllL}\ p\ l_1) \wedge (\mathsf{AllL}\ p\ l_2)$
5. $(\mathsf{AllL}\ (< x)\ l_1) \Rightarrow \mathsf{insert}\ x\ l_1 = x : l_1$

Proof. Simple induction on l_1.

Lemma 2. *For every tree t and value x,*

1. $(\mathsf{AllT}\ p\ t) \Rightarrow\ (\mathsf{AllL}\ p\ (\mathsf{BST2list}\ t))$
2. $(\mathsf{AllT}\ p\ t) \Rightarrow\ (\mathsf{AllL}\ p\ (\mathsf{H2list}\ t))$
3. $(p\ x) \wedge (\mathsf{AllT}\ p\ t) \Rightarrow (\mathsf{AllT}\ p\ (\mathsf{istBST}\ x\ t)$
4. $(p\ x) \wedge (\mathsf{AllT}\ p\ t) \Rightarrow (\mathsf{AllT}\ p\ (\mathsf{istH}\ x\ t)$

Proof. Induction on t.

We are now able to prove the required properties. For heapsort, we explore the fact that the intermediate structure is a heap (its root keeps the least element).

For the heapsort algorithm, we explore the fact that the intermediate tree is a heap.

Proposition 8.

$$(\mathsf{HEAP}\ t) \Rightarrow \mathsf{insert}\ x\ (\mathsf{H2list}\ t) = \mathsf{H2list}\ (\mathsf{istH}\ t)$$

Proof. By induction on the structure of t. The base case follows immediately from the definitions. For the induction step we have:

$$
\begin{array}{ll}
 & \mathsf{insert}\ x\ (\mathsf{H2list}\ (\mathsf{Node}\ y\ l\ r)) \\
= & \qquad \{\text{def. } \mathsf{H2list}\} \\
 & \mathsf{insert}\ x\ (y : (\mathsf{H2list}\ l) \odot (\mathsf{H2list}\ r)) \\
= & \qquad \{\text{def. } \mathsf{insert}\} \\
 & \begin{cases} x : y : ((\mathsf{H2list}\ l) \odot (\mathsf{H2list}\ r)) & \text{if } x < y, \\ y : (\mathsf{insert}\ x\ ((\mathsf{H2list}\ l) \odot (\mathsf{H2list}\ r))) & \text{if } x \geq y, \end{cases} \\
= & \qquad \{\text{lemma 1 (5)}\} \\
 & \begin{cases} x : (\mathsf{insert}\ y\ ((\mathsf{H2list}\ l) \odot (\mathsf{H2list}\ r))) & \text{if } x < y, \\ y : (\mathsf{insert}\ x\ ((\mathsf{H2list}\ l) \odot (\mathsf{H2list}\ r))) & \text{if } x \geq y, \end{cases} \\
= & \qquad \{\text{commutativity and associativity of } \odot\} \\
 & \begin{cases} x : (([y] \odot (\mathsf{H2list}\ r)) \odot (\mathsf{H2list}\ l)) & \text{if } x < y, \\ y : (([x] \odot (\mathsf{H2list}\ r)) \odot (\mathsf{H2list}\ l)) & \text{if } x \geq y, \end{cases} \\
= & \qquad \{\text{induction hypotheses}\} \\
 & \begin{cases} x : ((\mathsf{H2list}\ (\mathsf{istH}\ y\ l)) \odot (\mathsf{H2list}\ r)) & \text{if } x < y, \\ y : ((\mathsf{H2list}\ (\mathsf{istH}\ x\ l)) \odot (\mathsf{H2list}\ r)) & \text{if } x \geq y, \end{cases} \\
= & \qquad \{\text{def. } \mathsf{H2list}\} \\
 & \begin{cases} \mathsf{H2list}\ (\mathsf{Node}\ x\ (\mathsf{istH}\ y\ l)\ r) & \text{if } x < y, \\ \mathsf{H2list}\ (\mathsf{Node}\ y\ (\mathsf{istH}\ x\ l)\ r) & \text{if } x \geq y, \end{cases} \\
= & \qquad \{\text{def. } \mathsf{istH}\} \\
 & \mathsf{H2list}\ (\mathsf{istH}\ x\ (\mathsf{Node}\ y\ l\ r))
\end{array}
$$

To prove that the intermediate tree is actually a heap, we prove that insertion of elements preserves the invariant.

Proposition 9. *For every value x and tree t,*

$$(\mathsf{HEAP}\ t) \Rightarrow (\mathsf{HEAP}\ (\mathsf{istH}\ x\ t))).$$

Proof. Induction on t. The base case follows immediately from the definitions. For the induction step we have:

$$
\begin{array}{ll}
 & (\mathsf{HEAP}\ (\mathsf{istH}\ x\ (\mathsf{Node}\ y\ l\ r))) \\
\Leftrightarrow & \qquad \{\text{def. } \mathsf{istH}\} \\
 & \begin{cases} (\mathsf{HEAP}\ (\mathsf{Node}\ x\ (\mathsf{istH}\ y\ r)\ l)) & \text{if } x < y, \\ (\mathsf{HEAP}\ (\mathsf{Node}\ y\ (\mathsf{istH}\ x\ r)\ l)) & \text{if } x \geq y, \end{cases}
\end{array}
$$

In fact, when $x < y$ we have:

$$
\begin{cases}
(\mathsf{AllT}\ (\geq x)\ (\mathsf{istH}\ y\ l)) & \text{by lemma 2 (2) and } (\mathsf{HEAP}\ (\mathsf{Node}\ y\ l\ r)) \\
(\mathsf{AllT}\ (\geq x)\ r) & \text{by } (\mathsf{HEAP}\ (\mathsf{Node}\ y\ l\ r)) \\
(\mathsf{HEAP}\ (\mathsf{istH}\ x\ r)) & \text{by induction hypotheses and } (\mathsf{HEAP}\ (\mathsf{Node}\ y\ l\ r)) \\
(\mathsf{HEAP}\ l) & \text{by } (\mathsf{HEAP}\ (\mathsf{Node}\ y\ l\ r))
\end{cases}
$$

We reason similarly when $x \geq y$.

And now, the required result follows directly by induction.

Corollary 1. *For every list l,*

$$(\mathsf{HEAP}\ (\mathsf{buildH}\ l))$$

Proof. Simple induction on l.

For the quicksort algorithm, we explore the fact that the intermediate tree is a binary search tree.

Proposition 10. *For every value x and tree t,*

$$(\mathsf{BST}\ t) \Rightarrow \mathsf{insert}\ x\ (\mathsf{BST2list}\ t) = \mathsf{BST2list}\ (\mathsf{istBST}\ t)$$

Proof. By induction on the structure of t. The base case follows immediately from the definitions. For the induction step we have:

$$\begin{array}{ll}
 & \mathsf{insert}\ x\ (\mathsf{BST2list}\ (\mathsf{Node}\ y\ l\ r)) \\
= & \qquad \{\text{def. } \mathsf{BST2list}\} \\
 & \mathsf{insert}\ x\ ((\mathsf{BST2list}\ l) +\!\!+ [y] +\!\!+ (\mathsf{BST2list}\ r)) \\
= & \qquad \{\text{lemma 1 (1,2), 2 (1) and hypotheses } (\mathsf{BST}\ (\mathsf{Node}\ y\ l\ r))\} \\
 & \begin{cases} (\mathsf{insert}\ x\ (\mathsf{BST2list}\ l)) +\!\!+ [y] +\!\!+ (\mathsf{BST2list}\ r) & \text{if } x < y, \\ (\mathsf{BST2list}\ l) +\!\!+ [y] +\!\!+ (\mathsf{insert}\ x\ (\mathsf{BST2list}\ r)) & \text{if } x \geq y, \end{cases} \\
= & \qquad \{\text{induction hypotheses}\} \\
 & \begin{cases} (\mathsf{BST2list}\ (\mathsf{istBST}\ x\ l)) +\!\!+ [y] +\!\!+ (\mathsf{BST2list}\ r) & \text{if } x < y, \\ (\mathsf{BST2list}\ l) +\!\!+ [y] +\!\!+ (\mathsf{BST2list}\ (\mathsf{istBST}\ x\ r)) & \text{if } x \geq y, \end{cases} \\
= & \qquad \{\text{def. } \mathsf{BST2list}\} \\
 & \begin{cases} \mathsf{BST2list}\ (\mathsf{Node}\ y\ (\mathsf{istBST}\ x\ l)\ r) & \text{if } x < y, \\ \mathsf{BST2list}\ (\mathsf{Node}\ y\ l\ (\mathsf{istBST}\ x\ r)) & \text{if } x \geq y. \end{cases} \\
= & \qquad \{\text{def. } \mathsf{istBST}\} \\
 & \mathsf{BST2list}\ (\mathsf{istBST}\ x\ (\mathsf{Node}\ y\ l\ r))
\end{array}$$

Again, we note that the insertion function preserves the invariant.

Proposition 11. *For every value x and tree t,*

$$(\mathsf{BST}\ t) \Rightarrow (\mathsf{BST}\ (\mathsf{istBST}\ x\ t))).$$

Proof. Induction on t. The base case follows immediately from the definitions. For the induction step we have:

$$\begin{array}{ll}
 & (\mathsf{BST}\ (\mathsf{istBST}\ x\ (\mathsf{Node}\ y\ l\ r))) \\
\Leftrightarrow & \qquad \{\text{def. } \mathsf{istBST}\} \\
 & \begin{cases} (\mathsf{BST}\ (\mathsf{Node}\ y\ (\mathsf{istBST}\ x\ l)\ r)) & \text{if } x < y, \\ (\mathsf{BST}\ (\mathsf{Node}\ y\ l\ (\mathsf{istBST}\ x\ r))) & \text{if } x \geq y, \end{cases}
\end{array}$$

In fact, when $x < y$ we have:

$$\begin{cases}
(\mathsf{AllT}\ (< y)\ (\mathsf{istBST}\ x\ l)) & \text{by lemma 2 (2)} \\
(\mathsf{AllT}\ (\geq y)\ r) & \text{by } (\mathsf{BST}\ (\mathsf{Node}\ y\ l\ r)) \\
(\mathsf{BST}\ (\mathsf{istBST}\ x\ l)) & \text{by induction hypotheses and } (\mathsf{BST}\ (\mathsf{Node}\ y\ l\ r)) \\
(\mathsf{BST}\ r) & \text{by } (\mathsf{BST}\ (\mathsf{Node}\ y\ l\ r))
\end{cases}$$

We reason similarly when $x \geq y$.

And the required result follows directly by induction.

Corollary 2. *For every list l,*

$$(\mathsf{BST}\ (\mathsf{buildBST}\ l))$$

Proof. Simple induction on l.

C Alternative Derivation

In this appendix we present a slight variation on the strategy for deriving the sorting algorithms. This variation clarifies the role of the invariants on intermediate structures in the correctness argument of these algorithms.

When we compare the proof effort required to establish the correctness of the "sorting by insertion" algorithms, we note that there is significant difference between isortLT and the other two algorithms (isortH and isortBST). As explained in the main text, this is because the correctness for the last two algorithms depend on properties of the intermediate structure. However, we can explain that difference at a more abstract level — one might argue that isortLT is closer to the specification of a *generic insertion sort* presented at Section 4. To illustrate this point, let us recall the definition of these algorithms (we omit the definitions not relevant for this discussion):

$$
\begin{aligned}
\mathsf{isortLT} &= \mathsf{LT2list} \circ \mathsf{buildLT} \\
\mathsf{isortH} &= \mathsf{H2list} \circ \mathsf{buildH} \\
\mathsf{isortBST} &= \mathsf{BST2list} \circ \mathsf{buildBST} \\
\mathsf{LT2list} &= \mathsf{foldLTree}\ (\odot)\ t \\
&\quad \text{where } t\ \mathsf{Nothing} = [\,] \\
&\qquad\quad t\ (\mathsf{Just}\ x) = [x] \\
\mathsf{H2list} &= \mathsf{foldr}\ \mathsf{aux}\ [\,] \\
&\quad \text{where } \mathsf{aux}\ x\ l\ r = x : (l \odot r) \\
\mathsf{BST2list} &= \mathsf{foldBTree}\ \mathsf{aux}\ [\,] \\
&\quad \text{where } \mathsf{aux}\ x\ l\ r = l +\!\!+ (x : r)
\end{aligned}
$$

We observe that LT2list uses only $\odot$ to construct (non trivial) lists. On the other side, H2list and BST2list make use of other functions (namely (:) and (++)). That distinction makes the later two sensible to the ordering attributes of the intermediate tree.

Let us make one step back and define the following variants of isortH and isortBST algorithms:

$$
\begin{aligned}
\mathsf{isortH}' &= \mathsf{BT2list} \circ \mathsf{buildH} \\
\mathsf{isortBST}' &= \mathsf{BT2list} \circ \mathsf{buildBST} \\
\mathsf{BT2list} &= \mathsf{foldr}\ \mathsf{aux}\ [\,] \\
&\quad \text{where } \mathsf{aux}\ x\ l\ r = [x] \odot (l \odot r)
\end{aligned}
$$

Now, the conversion of binary trees into lists (BT2list) does not assume any ordering constrains on these trees. In fact, BT2list and LT2list should be read as two instances of the same polytypic function.

It is interesting to verify that, for these modified functions, the correctness argument is essentially the same as for isortLT.

Proposition 12. isortH′ *and* isortBST′ *are sort algorithms.*

Proof. We instantiate eqs. (2) and (3) for both functions. We set $\epsilon = \mathsf{Empty}$, and thus eq. (2) results directly from the definition. For eq. (4), we need to prove that for every binary tree t

and value x,

$$(\mathsf{BT2list} \circ (\mathsf{istH}\ x))\ t = ((\mathsf{insert}\ x) \circ \mathsf{BT2list})\ t$$
$$(\mathsf{BT2list} \circ (\mathsf{istBST}\ x))\ t = ((\mathsf{insert}\ x) \circ \mathsf{BT2list})\ t$$

These are proved by induction on the structure of t. We show the proof of the first one (the second is similar). The base case is trivial. For the induction step we have:

$$\begin{array}{ll} & \mathsf{BT2list}(\mathsf{istH}\ x\ (\mathsf{Node}\ y\ l\ r)) \\ = & \quad \{\text{def. istH}\} \\ & \begin{cases} (\mathsf{BT2list}(\mathsf{Node}\ x\ (\mathsf{istH}\ y\ r)\ l)) & \text{if } x < y, \\ (\mathsf{BT2list}(\mathsf{Node}\ y\ (\mathsf{istH}\ x\ r)\ l)) & \text{if } x \geq y, \end{cases} \\ = & \quad \{\text{def. BT2list}\} \\ & \begin{cases} [x] \odot ((\mathsf{BT2list}\ (\mathsf{istH}\ y\ r)) \odot (\mathsf{BT2list}\ l)) & \text{if } x < y, \\ [y] \odot ((\mathsf{BT2list}\ (\mathsf{istH}\ x\ r)) \odot (\mathsf{BT2list}\ l)) & \text{if } x \geq y, \end{cases} \\ = & \quad \{\text{induction hypotheses}\} \\ & \begin{cases} [x] \odot (([y] \odot (\mathsf{BT2list}\ r)) \odot (\mathsf{BT2list}\ l)) & \text{if } x < y, \\ [y] \odot (([x] \odot (\mathsf{BT2list}\ r)) \odot (\mathsf{BT2list}\ l)) \end{cases} \\ = & \quad \{\text{comutativity and associativity of } \odot\} \\ & [x] \odot ([y] \odot ((\mathsf{BT2list}\ l) \odot (\mathsf{BT2list}\ r))) \\ = & \quad \{\text{def. BT2list}\} \\ & [x] \odot (\mathsf{BT2list}\ (\mathsf{Node}\ y\ l\ r)) \end{array}$$

In order to refine isortH' and $\mathsf{isortBST}'$ to heap sort and quicksort, we should now proceed in two independent paths:

- to show that the construction of the intermediate tree can be performed co-inductively (i.e. buildH and buildBST are equal to unfoldhsort and unfoldqsort respectively);
- to show that the tree conversion into the resultant list can be simplified to their standard formulation (i.e. BT2list can be replaced by H2list for the heapsort and by BST2list for the quicksort).

The first point was performed in the main text (c.f. Propositions 5 and 7). The second is the one that should consider the ordering properties induced by the building process for each case — more precisely, one proves:

$$\mathsf{BT2list} \circ \mathsf{buildH} = \mathsf{H2list} \circ \mathsf{buildH}$$
$$\mathsf{BT2list} \circ \mathsf{buildBST} = \mathsf{BST2list} \circ \mathsf{buildBST}$$

As in appendix B, it is convenient to make explicit the structural invariants possessed by the intermediate structures in each case. That is,

$$\begin{array}{rcl} (\mathsf{HEAP}\ t) & \Longrightarrow & \mathsf{BT2list}\ t = \mathsf{H2list}\ t \\ (\mathsf{BST}\ t) & \Longrightarrow & \mathsf{BT2list}\ t = \mathsf{BST2list}\ t \end{array}$$

The proof require a simple lemma relating $\odot$ with ordering predicates.

Lemma 3. *For every*

1. $(\mathsf{AllL}\ (x \leq)\ l_1) \implies [x] \odot l_1 = x : l_1$
2. $(\mathsf{AllL}\ (x >)\ l_1) \implies l_1 \odot (x : l_2) = l_1 +\!\!+ (x : l_2)$
3. $(\mathsf{AllL}\ p\ l_1) \wedge (\mathsf{AllL}\ p\ l_2) \implies (\mathsf{AllL}\ p\ (l_1 \odot l_2))$

Proof. The first two are proved by simple induction on the structure of l_1. The third by mutual induction on l_1 and l_2.

Now, the required properties follow by simple induction. The base case is, in both cases, trivial. For the induction step, we have for H2list:

$$
\begin{array}{ll}
 & \mathsf{BT2list}\ (\mathsf{Node}\ x\ l\ r)) \\
= & \qquad \{\text{def. BT2list}\} \\
 & [x] \odot ((\mathsf{BT2list}\ l) \odot (\mathsf{BT2list}\ r)) \\
= & \qquad \{\text{induction hypotheses}\} \\
 & [x] \odot ((\mathsf{H2list}\ l) \odot (\mathsf{H2list}\ r)) \\
= & \qquad \{\text{def. of HEAP and lemma 3 (1,3)}\} \\
 & x : ((\mathsf{H2list}\ l) \odot (\mathsf{H2list}\ r)) \\
= & \qquad \{\text{def. H2list}\} \\
 & \mathsf{H2list}\ (Node\ x\ l\ r)
\end{array}
$$

and for BST2list:

$$
\begin{array}{ll}
 & \mathsf{BT2list}\ (\mathsf{Node}\ x\ l\ r)) \\
= & \qquad \{\text{def. BT2list}\} \\
 & [x] \odot ((\mathsf{BT2list}\ l) \odot (\mathsf{BT2list}\ r)) \\
= & \qquad \{\text{induction hypotheses}\} \\
 & [x] \odot ((\mathsf{BST2list}\ l) \odot (\mathsf{BST2list}\ r)) \\
= & \qquad \{\text{commutativity and associativity of } \odot\} \\
 & (BST2list\ l) \odot ([x] \odot (\mathsf{BST2list}\ r)) \\
= & \qquad \{\text{def. of BST and lemma 3 (1,2)}\} \\
 & (\mathsf{BST2list}\ l) \odot (x : (\mathsf{BST2list}\ r)) \\
= & \qquad \{\text{def. BST2list}\} \\
 & \mathsf{BST2list}\ (Node\ x\ l\ r)
\end{array}
$$