Verifying weight biased leftist heaps using dependent types

Jan Stolarek

Instytut Informatyki
Politechnika Łódzka
Poland
jan.stolarek@p.lodz.pl

Abstract. This paper is an intermediate level tutorial on verification using dependent types in Agda. It is also a case study of weight biased leftist heap data structure in a purely functional, dependently typed setting. Paper demonstrates how to write a formally verified implementation that is guaranteed to maintain that structure's invariants. The reader will learn how to construct complex equality proofs by building them from smaller parts. This knowledge will enable the reader to understand more advanced verification techniques, eg. equational reasoning provided by Agda's standard library or tactics system found in Idris and Coq programming languages.

1 Introduction

Formal verification is a subject that constantly attracts attention of the research community. Static type systems are considered to be a lightweight verification method but they can be very powerful and precise as well. Dependent type systems in languages like Agda [1], Idris [2] or Coq [3] can be successfully applied in practical verification tasks but they are not yet as widely used as they could potentially be. This paper contributes to changing that.

1.1 Motivation

Two things have motivated me to write this paper. Firstly, while there are many tutorials on dependently typed programming and basics of verification, I could find little material demonstrating how to put verification to practical use. A must-read introductory paper "Why Dependent Types Matter" by Altenkirch, McKinna and McBride [4], which demonstrates how to use dependent types to prove correctness of merge sort algorithm, actually elides many proof details that are required in a real-world application. I want to fill in that missing gap by writing a tutorial that picks up where other tutorials have ended.

My second motivation comes from reading Okasaki's classical "Purely Functional Data Structures" [5]. Despite book's title many presented implementations are not purely functional as they make use of impure exceptions to handle corner cases (eg. taking head of an empty list). I realised that using dependent types allows to do better and it is instructive to build a provably correct purely functional data structure on top of Okasaki's presentation.

In the end this paper is both a tutorial and a case study of weight biased leftist heap implemented in dependently typed setting. My goal is to teach the reader how to build complex proofs from simple ones. As a result the reader will be able to verify that operations on a data structure maintain required invariants. Acquired knowledge will allow to understand more advanced verification techniques, eg. equational reasoning provided by Agda's standard library or tactics system found in Idris [2] and Coq [3].

1.2 Companion code

This tutorial comes with a standalone companion code written in Agda 2.3.4[1][2]. I assume the reader is reading companion code along with the paper. Due to space limitations I elide some proofs that are detailed in the code using Notes convention adapted from GHC project [6].

"Living" version of companion code is available at GitHub[3] and it may receive updates after the paper is published.

1.3 Assumptions

I assume that reader has basic understanding of Agda, some elementary definitions and proofs. In particular I assume the reader is familiar with definition of natural numbers (`Nats`) and their addition (`+`) as well as proofs of basic properties of addition like associativity, commutativity or 0 as right identity ($a{+}0 \equiv a$). Reader should also understand `refl` with its basic properties (symmetry, congruence, transitivity and substitution), know the concept of "data as evidence" and other ideas presented in "Why Dependent Types Matter" [4] as I will build upon them. All of these are implemented in the `Basics` module in the companion code. Module `Basics.Reasoning` reviews in detail the above-mentioned proofs.

1.4 Notation and conventions

In the rest of the paper I will denote heaps using `typewriter font` and their ranks using an *italic type*. The description of merge algorithm will mention heaps `h1` and `h2` with ranks *h1* and *h2* respectively, their left children (`l1` in `h1` and `l2` in `h2`) and right children (`r1` in `h1` and `r2` in `h2`) with `p1` and `p2` as the priorities of root elements in `h1` and `h2` respectively. In the text I will use ⊕ to denote heap merging operation. So `h1`⊕`h2` will be a heap created by merging `h1` with `h2`, while *h1*⊕*h2* will be the rank of the merged heap.

[1] `http://ics.p.lodz.pl/~stolarek/_media/pl:research:dep-typed-wbl-heaps.tar.gz`

[2] NOTE TO THE REVIEWER: Agda 2.3.4 is planned to be released sometime in February or March. Companion code was created using latest development version of Agda. This notice will be removed from final version of the paper.

[3] `https://github.com/jstolarek/dep-typed-wbl-heaps`

I will represent priority using natural numbers with lower number meaning higher priority. This means that 0 will be the highest priority, while the lowest priority will be unbounded. This also means that if `p1 > p2` holds as a relation on natural numbers then `p2` is higher priority than `p1`.

In the text I will use numerals to represent `Nat`s but the code uses encoding based on `zero` and `suc`. Thus 2 in the text will correspond to `suc (suc zero)` in the source code.

I will use `{ }?` in code listings to represent Agda holes.

Remember that any sequence of Unicode characters is a valid identifier in Agda. Thus `l≥r` is an identifier, while `l ≥ r` is application of ≥ operator to `l` and `r` operands.

1.5 Contributions

This paper contributes the following:

- Section 3 presents unverified implementation and the problem of partiality of functions operating on a weight biased leftist heap. While the problem in general is well-known the solution to this particular case can be combined with verification of one of data structure's invariants. This is done in Section 4.
- Section 5 outlines a technique for constructing equality proofs using transitivity of propositional equality. This simple, standalone technique provides ground for understanding verification mechanisms used in Agda's standard library.
- Section 6 uses the technique introduced in Section 5 to prove code obtained by inlining one function into another. This shows how programs created from small, verified components can be proved correct by composing proofs of these components.
- Section 7 contains a case study of how a proof of data structure invariant influences the design of an API. This is demonstrated on the example of priority invariant proof and its influence on designing insertion of a new element into a heap.

2 Weight biased leftist heaps

A heap is a tree-based data structure used to implement priority queue. Each node in a heap satisfies *priority property*: priority of element at the node is not lower than priority of the children nodes[4]. Therefore element with the highest priority is stored at the root. Access to it has $O(1)$ complexity.

Weight biased leftist tree [7] is a binary tree that satisfies *rank property*: for each node rank of its left child is not smaller than rank of its right child. Rank of a tree is defined as its size (number of nodes). Weight biased leftist tree that satisfies priority property is called a weight biased leftist heap.

[4] I will also refer to children of a node as "subtrees" or "subheaps".

Right spine of a node is the rightmost path from that node to an empty node. From priority property it follows that right spine of a weight biased leftist heap is an ordered list (in fact, any path from root to a leaf is!). Two weight biased leftist heaps can be merged in $O(\log n)$ time by merging their right spines in the same way one merges ordered lists and then swapping children along the right spine of merged heap to restore rank property [5]. Inserting new element into weight biased leftist heap can be defined as merging existing heap with a newly created singleton heap (ie. a heap storing exactly one element). Deleting element with the highest priority can be defined as merging children of the root element.

3 Unverified implementation[5]

We begin by implementing the described algorithms without any proof of their correctness. We define `Heap` datatype as:

```
data Heap : Set where
  empty : Heap
  node  : Priority → Rank → Heap → Heap → Heap
```

According to this definition a heap is either empty or it is a node with priority, rank and two subheaps. Both `Priority` and `Rank` are aliases to `Nat`, which allows us to perform on them any operation that works on `Nat` type. Note that storing rank in a node is redundant: we could just compute size of a heap whenever necessary. I choose to store rank in the constructor because it will be instructive to show how it is converted into inductive type family index (see Section 4).

3.1 Merging two heaps

Heaps `h1` and `h2` are merged using a recursive algorithm. We need to consider four cases:

1. (base case) `h1` is empty: return `h2`.
2. (base case) `h2` is empty: return `h1`.
3. (inductive case) priority `p1` is higher than `p2`: `p1` becomes new root, `l1` becomes its one child and `r1⊕h2` becomes the other.
4. (inductive case) priority `p2` is higher than or equal to `p1`: `p2` becomes new root, `l2` becomes its one child and `r2⊕h1` becomes the other.

There is no guarantee that *r1*$\oplus$*h2* (or *r2*$\oplus$*h1*) is smaller than *l1* (or *l2*). To ensure that rank invariant is maintained we use helper function `makeT`, as proposed by Okasaki [5]. We pass new children and the priority to `makeT`, which creates a new node with the given priority and swaps the children if necessary (see Listing 1). As Okasaki points out this algorithm can be view as having two passes: a top-down pass that performs merging and a bottom-up pass that restores the rank invariant.

[5] Implementation for this section is located in the `TwoPassMerge.NoProofs` module of the companion code.

```
makeT : Priority → Heap → Heap → Heap
makeT p l r with rank l ≥ rank r
makeT p l r | true  = node p (suc (rank l + rank r)) l r
makeT p l r | false = node p (suc (rank l + rank r)) r l

merge : Heap → Heap → Heap
merge empty h2 = h2
merge h1 empty = h1
merge (node p1 h1-r l1 r1) (node p2 h2-r l2 r2)
  with p1 < p2
merge (node p1 h1-r l1 r1) (node p2 h2-r l2 r2)
  | true  = makeT p1 l1 (merge r1 (node p2 h2-r l2 r2))
merge (node p1 h1-r l1 r1) (node p2 h2-r l2 r2)
  | false = makeT p2 l2 (merge (node p1 h1-r l1 r1) r2)
```

Listing 1: Implementation of `makeT` and `merge`. `rank` returns rank of a tree.

3.2 Inserting element into a heap

Insert is defined by merging with a singleton heap as described in Section 2. See companion code for implementation.

3.3 Finding and removing element with the highest priority

To retrieve element with the highest priority we return value stored in the root of a heap:

```
findMin : Heap → Priority
findMin empty          = { }?
findMin (node p _ _ _) = p
```

Here we encounter a problem: what should `findMin` return for an empty heap? If we were using a language like Haskell or ML one thing we could consider is raising an exception. This is the choice made by Okasaki in "Purely Functional Data Structures". But throwing an exception is precisely the thing that would make our implementation impure! Besides Agda is a total language, which means that every function must terminate with a result. Raising an exception is therefore not an option. Another alternative is to assume a default priority that will be returned for an empty heap. This priority would have to be some distinguished natural number. 0 represents the highest priority so it is unreasonable to assume it as default. We could return ∞, which represents the lowest possible priority. This would require us to extend definition of `Nat` with ∞, which in turn would force us to modify all functions that pattern match on values of `Nat`. Redefining natural numbers for the sake of getting one function right also does not sound like a good option. Let's face it – our types do not reflect the fact that `findMin` function is not defined for an empty heap! To solve this problem we need to be more specific about types. One solution would be to use `Maybe` datatype:

```
data Maybe (A : Set) : Set where
  nothing : Maybe A
  just    : A → Maybe A

findMinM : Heap → Maybe Priority
findMinM empty          = nothing
findMinM (node p _ _ _) = just p
```

Returning `nothing` is like saying "no output exists for the given input data". This allows us to express the fact that `findMin` is not defined for some input values. This solution works but it forces every caller of `findMinM` to inspect the result and be prepared for `nothing`, which means extra boilerplate in the code and checks during run time. Implementation of `deleteMin` based on description in Section 2 faces the same problem.

The best solution to this issue is to ensure that `findMin` and `deleteMin` cannot be applied to an empty heap. We can achieve this by indexing `Heap` with its size. Doing so will also allow us to prove the rank property.

4 Proving rank property[6]

We will now prove that our implementation maintains the rank property. The first step is to express `Rank` at the type level as an index of `Heap` datatype. Since rank of a heap is now part of its type we can ensure at compile time that rank of left subtree is not smaller than rank of the right subtree. We do this be requiring that `node` constructor is given a proof that rank invariant holds. To express such proof we use ≥ datatype:

```
data _≥_ : Nat → Nat → Set where
  ge0 : {y : Nat} → y ≥ zero
  geS : {x y : Nat} → x ≥ y → suc x ≥ suc y
```

Values of this type, which is indexed by two natural numbers, prove that: a) any natural number is greater than or equal to `0` (`ge0` constructor); b) if two numbers are in greater-equal relation then their successors are also in that relation (`geS` constructor). This type represents concept of data as evidence [4]. We use `order` function to compare two natural numbers and `Order` datatype to express the result. Implementation is located in `Basics.Ordering` module of the companion code.

Having defined ≥ we can now give new definition of `Heap`:

```
data Heap : Rank → Set where
  empty : Heap zero
  node  : {l r : Rank} → Priority → l ≥ r →
          Heap l → Heap r → Heap (suc (l + r))
```

[6] Implementation for this section is located in the `TwoPassMerge.RankProof` module of the companion code.

Empty heap contains no elements and so `empty` returns `Heap` indexed with `0`. Non-empty node stores an element and two children of rank l and r. Therefore the size of the resulting heap is $1 + l + r$, which we express as $\mathrm{suc}(l + r)$. We must also supply a value of type `l ≥ r` to the constructor, ie. we must provide evidence that rank invariant holds.

Proving the rank invariant itself is surprisingly simple. We can obtain evidence that rank of left subtree is not smaller than rank of right subtree by replacing $\geq$ in `makeT` with `order`, which compares two `Nat`s and supplies evidence of the result. But there is another difficulty here. Recall that the merging algorithm is two pass: we use `merge` to actually do the merging and `makeT` to restore the rank invariant if necessary. Since we index heaps by their rank we now require that `makeT` and `merge` construct trees of correct rank. We must therefore prove that: a) `makeT` creates a node with rank equal to sum of children nodes' ranks plus one; b) `merge` creates a heap with rank equal to the sum of ranks of heaps being merged.

4.1 Proving makeT

`makeT` takes subtrees of rank l and r and produces a new tree with rank $\mathrm{suc}(l+r)$, where suc follows from the fact that the node itself is storing one element. We must prove that each of two cases of `makeT` returns heap of correct rank:

1. If l is greater than or equal to r then no extra proof is necessary as everything follows from the definition of $+$ and type signature of `node`.
2. If r is greater than or equal to l then we must swap `l` and `r` subtrees. This requires us to prove that:

   $$suc(r + l) \equiv suc(l + r)$$

   That proof is done using congruence on suc function and commutativity of addition. We will define that proof as `makeT-lemma`.

Listing 2 shows new code of `makeT`. Notice how `subst` applies the proof to the `Heap` type constructor and converts the type produced by `(node p r≥l r l)` expression into the type given in `makeT` type signature.

```
makeT-lemma : (a b : Nat) → suc (a + b) ≡ suc (b + a)
makeT-lemma a b = cong suc (+comm a b)

makeT : {l r : Rank} → Priority → Heap l → Heap r → Heap (suc (l + r))
makeT {l-rank} {r-rank} p l r with order l-rank r-rank
makeT {l-rank} {r-rank} p l r | ge l≥r
  = node p l≥r l r
makeT {l-rank} {r-rank} p l r | le r≥l
  = subst Heap (makeT-lemma r-rank l-rank) (node p r≥l r l)
```

Listing 2: Implementation of `makeT` with verified rank property.

4.2 Proving merge

We now verify that all four cases of `merge` shown in Listing 1 produce heap of required rank.

base cases In the first base case we have $h1 \equiv 0$. Therefore:

$$h1 + h2 \equiv 0 + h2 \overset{+,(1)}{\equiv} h2$$

Which ends the first proof – everything follows from definition of +[7]. In the second base case $h2 \equiv 0$ and things are slightly more difficult: the definition of + only says that 0 is the left identity but it does not say that it is also the right identity. We must therefore construct a proof that:

$$h1 + 0 \overset{?}{\equiv} h1$$

Luckily for us, we already have that proof defined in the `Basics.Reasoning` module as `+0`. Since that proof is in the opposite direction – it proves $a \equiv a + 0$, not $a + 0 \equiv a$ – we have to use symmetry of $\equiv$.

inductive cases In an inductive case we know that neither `h1` nor `h2` is empty, ie. their ranks are given as $\mathrm{suc}(l1+r1)$ and $\mathrm{suc}(l2+r2)$ respectively. This means that Agda sees expected rank of the merged heap as:

$$\mathrm{suc}(l1 + r1) + \mathrm{suc}(l2 + r2) \overset{+,(2)}{\equiv} \mathrm{suc}((l1 + r1) + \mathrm{suc}(l2 + r2))$$

This will be our goal in both proofs of inductive cases.

In the first inductive case we construct the result by calling[8]:

```
makeT p1 l1 (merge r1 (node p2 l2≥r2 l2 r2))
```

Call to `node` with `l2` and `r2` as parameters produces node of rank $\mathrm{suc}(l2 + r2)$. Passing it to `merge` together with `r1` gives a tree of rank $r1 + \mathrm{suc}(l2 + r2)$ (by the type signature of `merge`). Passing result of `merge` to `makeT` produces tree of rank $\mathrm{suc}(l1 + (r1 + \mathrm{suc}(l2 + r2)))$ by the type signature of `makeT`. We must therefore construct a proof that:

$$\mathrm{suc}(l1 + (r1 + \mathrm{suc}(l2 + r2))) \equiv \mathrm{suc}((l1 + r1) + \mathrm{suc}(l2 + r2))$$

Appealing to congruence on suc leaves us with a proof of:

$$l1 + (r1 + \mathrm{suc}(l2 + r2)) \equiv (l1 + r1) + \mathrm{suc}(l2 + r2)$$

---

[7] The $\overset{+,(1)}{\equiv}$ notation means that equality follows from the first defining equation of +.

[8] Note that `node` constructor in the unverified implementation show in Listing 1 takes slightly different parameters. This is because we changed the definition of `Heap` datatype to take the proof of rank property instead of storing the rank in the constructor.

Substituting $a = l1$, $b = r1$ and $c = \mathrm{suc}(l2 + r2)$ gives:

$$a + (b + c) \equiv (a + b) + c$$

This is associativity of addition that we have already proved in `Basics.Reasoning`.

The proof of second inductive case is much more interesting. This time we construct the result by calling:

```
makeT p2 l2 (merge r2 (node p1 l1≥r1 l1 r1))
```

and therefore have to prove:

$$\mathrm{suc}(l2 + (r2 + \mathrm{suc}(l1 + r1))) \equiv \mathrm{suc}((l1 + r1) + \mathrm{suc}(l2 + r2))$$

Again we use congruence to deal with the outer calls to suc and substitute $a = l2$, $b = r2$ and $c = l1 + r1$. This leaves us with a proof of lemma A:

$$a + (b + \mathrm{suc}\, c) \equiv c + \mathrm{suc}(a + b)$$

From associativity we know that:

$$a + (b + \mathrm{suc}\, c) \equiv (a + b) + \mathrm{suc}\, c$$

If we prove lemma B:

$$(a + b) + \mathrm{suc}\, c \equiv c + \mathrm{suc}(a + b)$$

then we can combine lemmas A and B using transitivity to get the final proof. We substitute $n = a + b$, $m = c$ and rewrite lemma B as:

$$n + \mathrm{suc}\, m \equiv m + \mathrm{suc}\, n$$

From symmetry of `+suc` we know that:

$$n + \mathrm{suc}\, m \equiv \mathrm{suc}(n + m)$$

Using transitivity we combine it with congruence on commutativity of addition to prove:

$$\mathrm{suc}(n + m) \equiv \mathrm{suc}(m + n)$$

Again, using transitivity we combine it with `+suc` to show:

$$\mathrm{suc}(m + n) \equiv m + \mathrm{suc}\, n$$

Which proves lemma B and therefore the whole proof is complete (Listing 3, see companion code for complete implementation).

```
lemma-B : (n m : Nat) → n + suc m ≡ m + suc n
lemma-B n m = trans (sym (+suc n m)) (trans (cong suc (+comm n m)) (+suc m n))

lemma-A : (a b c : Nat) → a + (b + suc c) ≡ c + suc (a + b)
lemma-A a b c = trans (+assoc a b (suc c)) (lemma-B (a + b) c)

proof-2 : (l1 r1 l2 r2 : Nat) → suc (l2 + (r2  + suc (l1 + r1)))
                              ≡ suc ((l1 + r1) + suc (l2 + r2))
proof-2 l1 r1 l2 r2 = cong suc (lemma-A l2 r2 (l1 + r1))
```

Listing 3: Proof of second inductive case of `merge`.

4.3 insert

Inserting new element into the heap increases its rank by one. Now that rank is encoded as a datatype index this fact must be reflected in the type signature of `insert`. As previously we define `insert` as `merge` with a singleton heap. Rank of singleton heap is 1 (ie. `suc zero`), while already existing heap has rank n. According to definition of merge the resulting heap will therefore have rank:

$$(\mathrm{suc}\,\mathrm{zero}) + n \overset{+,(2)}{\equiv} \mathrm{suc}(\mathrm{zero} + n) \overset{+,(1)}{\equiv} \mathrm{suc}\, n$$

Which is the size we require in the type signature of `insert`. This means we don't need any additional proof because expected result follows from definition.

4.4 findMin, deleteMin

Encoding rank at the type level allows us to write total versions of `findMin` and `deleteMin`. By requiring that input `Heap` has rank `suc n` we exclude the possibility of passing empty heap to any of these functions.

5 Constructing equality proofs using transitivity

Now that we have conducted an inductive proof of `merge` in Section 4.2 we can focus on a general technique used in that proof. Let us rewrite `proof-2` in a different way to see closely what is happening at each step. Inlining lemmas A and B into `proof-2` gives:

```
proof-2i : (l1 r1 l2 r2 : Nat) → suc (l2 + (r2  + suc (l1 + r1)))
                               ≡ suc ((l1 + r1) + suc (l2 + r2))
proof-2i l1 r1 l2 r2 =
  cong suc (trans (+assoc l2 r2 (suc (l1 + r1)))
           (trans (sym (+suc (l2 + r2) (l1 + r1)))
           (trans (cong suc (+comm (l2 + r2) (l1 + r1)))
                  (+suc (l1 + r1) (l2 + r2)))))
```

We see that `proof-2` is structured around proofs of elementary properties combined using transitivity. In general, if we have to prove $a \equiv e$ and we can prove $a \equiv b$ using proof 1, $b \equiv c$ using proof 2, $c \equiv d$ using proof 3, $d \equiv e$ using proof 4 then we can combine these proofs to get the final proof of $a \equiv e$:

$$\text{trans}\,(\text{proof 1})\,(\text{trans}\,(\text{proof 2})\,(\text{trans}\,(\text{proof 3})\,(\text{proof 4})))$$

While simple to use, combining proofs using transitivity can be hard to comprehend. The intermediate terms are hidden from us and we have to reconstruct them every time we read our proof. Let us then replace usage of transitivity with the following notation, which explicitly shows intermediate proof steps together with their proofs:

$$\begin{aligned}
a &\equiv \langle \text{proof 1} \rangle \\
b &\equiv \langle \text{proof 2} \rangle \\
c &\equiv \langle \text{proof 3} \rangle \\
d &\equiv \langle \text{proof 4} \rangle \\
e &
\end{aligned}$$

Rewriting `proof-2i` in this notation gives:

$$\begin{aligned}
\text{suc}(l2 + (r2 + \text{suc}(l1 + r1))) &\equiv \langle \text{cong suc} \rangle \\
\text{suc}(l2 + (r2 + \text{suc}(l1 + r1))) &\equiv \langle \text{+assoc } l2\ r2\ (\text{suc}(l1 + r1)) \rangle \\
\text{suc}((l2 + r2) + \text{suc}(l1 + r1)) &\equiv \langle \text{sym}(\text{+suc } (l2 + r2)\ (l1 + r1)) \rangle \\
\text{suc}(\,\text{suc}((l2 + r2) + (l1 + r1))) &\equiv \langle \text{cong suc } (\text{+comm } (l2 + r2)\ (l1 + r1)) \rangle \\
\text{suc}(\,\text{suc}((l1 + r1) + (l2 + r2))) &\equiv \langle \text{+suc } (l1 + r1)\ (l2 + r2) \rangle \\
\text{suc}((l1 + r1) + \text{suc}(l2 + r2)) &
\end{aligned}$$

Grey suc denotes that everything happens under call to `suc` (thanks to using congruence on suc as the first proof). Comparing this notation to `proof-2i` on the previous page shows that proofs in angle brackets correspond to proofs combined using trans, while series of expressions left of $\equiv$ parallels our reasoning from Section 4.2.

6 Proving rank property for single pass merge by composing existing proofs[9]

As mentioned in Section 3.1 `merge` can be viewed as consisting of two passes. We can obtain a single pass version of the algorithm by inlining calls to `makeT` into `merge`. This new algorithm has two bases cases (as previously) and four inductive cases:

[9] Implementation for this section is located in the `SinglePassMerge.RankProof` module of the companion code.

1. (base case) `h1` is empty: return `h2`.
2. (base case) `h2` is empty: return `h1`.
3. (1st inductive case) priority `p1` is higher than `p2` and *l1* is not smaller than *r1*$\oplus$*h2*: `p1` becomes new root, `l1` becomes the left child and `r1`$\oplus$`h2` becomes the right child.
4. (2nd inductive case) priority `p1` is higher than `p2` and *r1*$\oplus$*h2* is larger than *l1*: `p1` becomes new root, `r1`$\oplus$`h2` becomes the left child and `l1` becomes the right child.
5. (3rd inductive case) priority `p2` is higher than or equal to `p1` and *l2* is not smaller than *r2*$\oplus$*h1*: `p2` becomes new root, `l2` becomes the left child and `r2`$\oplus$`h1` becomes the right child.
6. (4th inductive case) priority `p2` is higher than or equal to `p1` and *r2*$\oplus$*h1* is larger than *l2*: `p2` becomes new root, `r2`$\oplus$`h1` becomes the left child and `l2` becomes the right child.

Now that we have inlined `makeT` we must construct proofs of new `merge`. Note that previously we made calls to `makeT` only in inductive cases. This means that implementation of base cases remains unchanged and so do the proofs. Let us take a closer look at proofs we need to supply for inductive cases:

– (1st inductive case): call to `makeT` would not swap left and right children when creating a node from parameters passed to it. We must prove:

$$\mathrm{suc}(l1 + (r1 + \mathrm{suc}(l2 + r2))) \equiv \mathrm{suc}((l1 + r1) + \mathrm{suc}(l2 + r2))$$

– (2nd inductive case): call to `makeT` would swap left and right children when creating a node from parameters passed to it. We must prove:

$$\mathrm{suc}((r1 + \mathrm{suc}(l2 + r2)) + l1) \equiv \mathrm{suc}((l1 + r1) + \mathrm{suc}(l2 + r2))$$

– (3rd inductive case): call to `makeT` would not swap left and right children when creating a node from parameters passed to it. We must prove:

$$\mathrm{suc}(l2 + (r2 + \mathrm{suc}(l1 + r1))) \equiv \mathrm{suc}((l1 + r1) + \mathrm{suc}(l2 + r2))$$

– (4th inductive case): call to `makeT` would swap left and right children when creating a node from parameters passed to it. We must prove:

$$\mathrm{suc}((r2 + \mathrm{suc}(l1 + r1)) + l2) \equiv \mathrm{suc}((l1 + r1) + \mathrm{suc}(l2 + r2))$$

First thing to note is that inductive cases 1 and 3 require us to supply the same proofs as the ones we gave for inductive cases in two-pass merge. This means we can reuse old proofs. What about cases 2 and 4? One thing we could do is construct proofs of these properties from scratch using technique described in Section 5. This is left as an exercise to the reader. Here we will proceed in a different way.

Notice that properties we have to prove in cases 2 and 4 are very similar to properties 1 and 3. The only difference between 1 and 2 and between 3 and 4 is the order of parameters inside outer suc on the left hand side of equality. This is expected: in cases 2 and 4 we swap left and right subtree passed to `node` and this is directly reflected in the types. Now, if we could prove that:

$$\mathrm{suc}((r1 + \mathrm{suc}(l2 + r2)) + l1) \equiv \mathrm{suc}(l1 + (r1 + \mathrm{suc}(l2 + r2)))$$

and

$$\mathrm{suc}((r2 + \mathrm{suc}(l1 + r1)) + l2) \equiv \mathrm{suc}(l2 + (r2 + \mathrm{suc}(l1 + r1)))$$

then we could use transitivity to combine these proofs with proofs of inductive cases 1 and 3. If we abstract the parameters in the above equalities we see that the property we need to prove in both cases is:

$$\mathrm{suc}(a + b) \equiv \mathrm{suc}(b + a)$$

And that happens to be `makeT-lemma` from Section 4.1! New version of `merge` was created by inlining calls to `makeT` and now it turns out we can construct proofs of that implementation by composing proofs of `makeT` and `merge` using transitivity. This is exactly the same technique that we developed in Section 5 only this time it is used on a slightly larger scale. It leads to a very elegant solution presented in module `SinglePassMerge.RankProof` of the companion code.

7 Proving priority property[10]

To prove priority property I will index `Heap` with `Priority` and use technique demonstrated by Altenkirch, McBride and McKinna in Section 5.2 of "Why Dependent Types Matter" [4][11]. Index of value n says that "this heap can store elements with priorities n or lower". In other words `Heap` indexed with 0 can store any priority, while `Heap` indexed with 3 can store priorities 3, 4 and lower, but can't store 0, 1 and 2. The new definition of `Heap` looks like this[12]:

```
data Heap : Priority → Set where
  empty : {n : Priority} → Heap n
  node  : {n : Priority} → (p : Priority) → Rank → p ≥ n →
          Heap p → Heap p → Heap n
```

As always `Heap` has two constructors. The `empty` constructor returns `Heap n`, where index n is not constrained in any way. This means that empty heap can be given any restriction on priorities of stored elements. The `node` constructor

[10] Implementation for this section is located in the `TwoPassMerge.PriorityProof` module of the companion code.

[11] To keep things simple let's forget about rank proof we conducted earlier – in this section we once again store rank explicitly in the `node` constructor.

[12] Actual implementation in the companion code is slightly different. It uses sized types [8] to guide the termination checker in the `merge` function. This issue is orthogonal to the conducted proofs, hence I avoid sized types here for the sake of simplicity.

also creates `Heap n` but this time `n` is constrained. If we store priority `p` in a node then:

1. the resulting heap can only be restricted to store priorities at least as high as `p`. For example, if we create a node that stores priority 3 we cannot restrict the resulting heap to store priorities 4 and lower, because the fact that we store 3 in that node violates the restriction. This restriction is expressed by the `p ≥ n` parameter: if we can construct a value of type `p ≥ n` then it becomes a proof that priority `p` is lower than or equal to `n`.
2. children of a node can only be restricted to store priorities that are not higher than `p`. Example: if we restrict a node to store priorities 4 and lower we cannot restrict its children to store priorities 3 or higher. This restriction is expressed by index `p` in the subheaps passed to node constructor.

Altenkirch, McKinna and McBride [4] used this technique to prove correctness of merge sort for lists. In a weight biased leftist heap every path from root to a leaf is a sorted list so extending their technique to heap case is straightforward. I elide discussion of `merge` as I offer nothing new compared to Altenkirch's paper. I instead focus on issue of creating singleton heaps and inserting elements into a heap as these cases now become interesting.

When creating a singleton heap we have to answer a question: "what priorities can we later store in a singleton heap that we just created?". "Any" seems to be a reasonable answer, which means the resulting heap will be indexed with 0 meaning: "Priorities lower than or equal to 0 – ie. any priorities – can be stored in this Heap". With such a liberal definition of singleton heap it is easy to write definition of `insert` by requiring that both input and output heap can store any priorities:

```
singleton : (p : Priority) → Heap zero
singleton p = node p (suc zero) ge0 empty empty

insert : Priority → Heap zero → Heap zero
insert p h = merge (singleton p) h
```

But what if we want to insert into a heap that is not indexed with 0? One solution is to be liberal and "promote" that heap so that after insertion it can store elements with any priorities. Priority restriction can always be loosened but it cannot be tightened easily. However such a liberal approach might not always be satisfactory. We might actually want to keep priority bounds as tight as possible. Let us explore that possibility.

We begin by rewriting the `singleton` function:

```
singletonB' : {b : Priority} → (p : Priority) → p ≥ b → Heap b
singletonB' p p≥b = node p one p≥b empty empty
```

Now `singletonB'` allows to construct a heap containing a single element with priority `p` but the whole heap is bounded by some `b`. To construct such a heap we must supply a proof that `p` can actually be stored in `Heap b`. We can now implement new insertion function:

```
insertB' : {b : Priority} → (p : Priority) → p ≥ b → Heap p → Heap b
insertB' p p≥b h = merge (singletonB' p p≥b) (liftBound p≥b h)
```

where `liftBound` is a function that loosens the priority bound of a heap given evidence that it is possible to do so (ie. that the new bound is less restrictive than the old one). But if we try to construct a heap using `insertB'` we quickly discover that it is useless:

```
example-heap : Heap zero
example-heap = (insertB' (suc zero) ge0
               (insertB' zero { }? empty))
```

In the second call to `insertB'` we are required to supply a proof that $0 \geq 1$, which of course is not possible. The problem is that using the new `insertB'` function we can only lower the bound on the heap and thus insert the elements into the heap in decreasing order:

```
example-heap : Heap zero
example-heap = (insertB' zero ge0
               (insertB' (suc zero) ge0 empty))
```

This is a direct consequence of our requirement that the heap we are inserting into is restricted exactly by the priority of element we are inserting.

The bottom line is: one has to carefully consider implications of a proof on the design of functions that manipulate the data structure.

8 Summary and further reading

This paper demonstrated a simple technique for building complex equality proofs by composing elementary equalities using transitivity. Understanding this approach allows the reader to learn more advanced techniques offered by dependently typed languages. Possible next steps include learning mechanisms offered by Agda's standard library, namely the `≡-Reasoning` located in `Relation.Binary.PropositionalEquality` module, which provides notation identical to the one introduced in Section 5. Another step to take is learning how to conduct proofs using tactics in languages such as Idris [2] or Coq [3].

Due to space limitations some parts of the companion code were left out from the discussion. The reader may now wish to take a look at verification of priority invariant in two-pass and single-pass implementations of `merge` and simultaneous proofs of rank and priority invariants.

Acknowledgements

I thank Andreas Abel for his help with using the sized types to guide Agda's termination checker.

References

1. Norell, U.: Towards a practical programming language based on dependent type theory. PhD thesis, Chalmers University of Technology (2007)
2. Brady, E.: Idris, a general-purpose dependently typed programming language: Design and implementation. Journal of Functional Programming **23**(5) (September 2013) 552–593
3. Coq development team development team development team development team development team development team development team development team development team development team development team development team development team development team development team development team development team: The Coq proof assistant reference manual. `http://coq.inria.fr`
4. Altenkirch, T., McBride, C., McKinna, J.: Why dependent types matter. Unpublished (2005)
5. Okasaki, C.: Purely functional data structures. Cambridge University Press (1999)
6. Marlow, S., Peyton Jones, S.: The Glasgow Haskell Compiler. In Brown, A., Wilson, G., eds.: The Architecture of Open Source Applications. Volume II. (2012)
7. Cho, S., Sahni, S.: Weight biased leftist trees and modified skip lists. Journal of Experimetnal Algorithmics **3** (1996) 361–370
8. Abel, A.: Semi-continuous sized types and termination. Logical Methods in Computer Science **4**(2) (2008)