Heap sort in Agda

Agda

Agda is a dependently typed programming language based on the Unified Theory of Dependent Types [1]. Being dependently typed can be understood simply as having values in the type definitions of programs. Using this feature, we can use the Curry-Howard correspondence [2] (i.e. propositions as types) to construct formal proofs for programs.

Let us examine the proof for the transitive property of inequality for natural numbers (Figure 1).

```
transitive : ∀ {x y z} → x ≤ y → y ≤ z → x ≤ z
transitive (zero≤ .0) (zero≤ z) = zero≤ z
transitive (zero≤ .(suc y)) (suc≤suc y z _) = zero≤ (suc z)
transitive (suc≤suc x y p) (suc≤suc .y z q) = suc≤suc x z (transitive p q)
```

Figure 1 Transitive property

This states that for any x, y, and z, if x ≤ y and y ≤ z then x ≤ z. To prove this, we use induction: if x and y is 0 then the proof is trivial. The next case is when x is 0, and y is non-zero; in this case the proof is still trivial since x is 0. In the final case when all variables are non-zero, we just need to reuse the previous cases to prove the proposition.

Leftist Heaps

A leftist heap is a binary tree data structure that holds the following properties:

- *Heap property*: The children of the tree must contain values no less than the parent. This ensures that the smallest element of the tree is always at the root, allowing $O(1)$ access.
- *Leftist property*: The right child of a tree must have a rank no more than the left child. The rank of the tree is defined as the shortest path to a leaf. This ensures that the right spine of the tree is the shortest path to a leaf.

If these properties are maintained then it results in an (intentionally) unbalanced binary tree, where the right spine is much shorter than the left. The merge operation can exploit this, by merging on the right spine, which is guaranteed to be the shortest path to merge on the tree. It can be shown that merge has a runtime complexity of $O(\log n)$, using this technique. The tree's right spine is longest when the tree is balanced. In this case the length of the right spine is $\log n$, therefore inserting a node along this spine takes at most $\log n$ comparisons.

Using induction, properties of trees can be proven too. The inductive definition of trees in Agda can be seen in Figure 2, as well as the two properties that need to be proven.

```
data Tree : (A : Set) → Set where
  leaf : ∀ {A} → Tree A
  branch : ∀ {A} → (l : Tree A) → (e : A) → (r : Tree A) → Tree A

data _IsHeap : (t : HTree) → Set where
  leafIsHeap : {t : HTree} → t ≡ leaf → t IsHeap
  branchIsHeap : ∀ {l i r} → {t : HTree} → t ≡ (branch l i r)
    → l IsHeap → r IsHeap
    → Item.value i ≤ value l → Item.value i ≤ value r
    → t IsHeap

data _IsLeftist : (t : HTree) → Set where
  leafIsLeftist : {t : HTree} → t ≡ leaf → t IsLeftist
  branchIsLeftist : ∀ {l i r} → {t : HTree} → t ≡ (branch l i r)
    → l IsLeftist → r IsLeftist
    → rank r ≤ rank l
    → rank t ≡ suc (rank r)
    → t IsLeftist
```

Figure 2 Tree, Heap and Leftist property in Agda

A `Tree` consists of either a leaf, or a branch that contains other trees. `HTree` is a `Tree` that contains a number value as well as a rank. `IsHeap` states that if a branch has two subtrees that are also heaps then if its value is no more than its children then it is also a heap. `IsLeftist` states that a branch has to have two subtrees that are also leftist, and its subtrees must be ordered so that the left one has a higher rank. The rank of the parent is also defined to be one larger than the right child's (smaller) rank.

Merge maintains these properties in two steps. In the first step it orders the two trees based on their value to keep the heap property. Since the two subtrees of a branch can be interchanged, and still keep the heap property, this can be used to maintain the leftist property. The second step does this exactly: if the merge of the remaining subtrees results in a tree that would break the leftist property then it swaps the left and right child of the tree. To see this algorithm in action see [3].

HEAP SORT

Heap sort relies on the heap data structure to sort lists. Its operation is simple, first turn the list into a heap (by repeatedly inserting the values into the tree as singletons), and then repeatedly removing the top of the heap to create an ordered list. Since insert and merge both have $O(\log n)$ complexity, therefore heap sort has an $O(n \log n)$ complexity.

To prove the correctness of this sorting algorithm, two properties need to be shown: that the resulting list is in fact ordered, and that the resulting list is a permutation of the input list. The second property is required, as the function that returns an empty list given any list, always returns an ordered list, but it cannot be said that it sorts the input list.

To prove that a list is ordered, one needs to show for every element in the list that it is no larger than any element that follow. This can be shown for flatten in two steps. First it needs to be shown that if a value is known to be less than the value of the heap being flattened, then the known value will be no larger than the values in the list the heap was flattened to. This is a simple proof using the transitive property of inequality, and the fact that this carries with merge. Then using this property, it can be shown that flatten produces an ordered list, as it exploits the heap property of the tree.

The flatten operation uses merging subtrees to reduce the size of the tree. Unfortunately, this is not structural recursion, therefore, the termination checker cannot prove that flatten terminates. The proof could be improved by introducing sized types to the tree structure, so that the termination checker understands that merging subtrees in fact reduces the tree's size.

To prove that a list is a permutation of another, then one needs to show that every element of one list is contained in the other and vice versa. To show that sort produces a permutation, it must be proven that the heap produced from the list contains all values from the list, and then that the flattened heap contains every value from the heap. This proof is rather tedious as the exact location of each value needs to be shown in each data structure (Tree, Heap, and List), to be complete.

In Agda this was shown only one direction, that is, the sorted list contains all elements of the original list. This is a lesser proof, as the list of all natural numbers could satisfy this property, given any list. The fact that natural numbers are non-unique causes some inconsistencies as well. For a stronger proof it should be shown that the sorted list is the original list with the elements inserted in a different order.

Conclusion

In the accompanying Agda source it is shown that a specific merge operation maintains the heap and leftist properties of a tree of natural numbers. This data structure was used to implement a sort operation over lists of natural numbers. It was shown that this operation in fact results in an ordered list. It was also proved that the sorted list contains all elements of the original list. In the future, this implementation should be generalised to types other than natural numbers. Furthermore, the proof for the sorted list being a permutation should be revised to be more robust.

References

[1] Z. Luo, *Computation and reasoning: a type theory for computer science*. Oxford University Press, Inc., 1994.

[2] W. A. Howard, 'The formulae-as-types notion of construction', *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, vol. 44, pp. 479–490, 1980.

[3] 'Leftist Heap Visualization'. [Online]. Available: https://www.cs.usfca.edu/~galles/visualization/LeftistHeap.html. [Accessed: 13-May-2018].