Verification Challenge, On Building Trees of Minimum Height

Liewe Thomas van Binsbergen   João Paulo Pizani Flor

July 5th, 2013

Accompanied files:

- Main.v (containing the main function and proof)
- SInc.v (containing the proves needed for $fold\_rightstep[]$)
- StepN.v (containg the definition of *step* using an amortizing argument)
- Minimum.v (containing the proves about $foldl1join$)
- Tree.v (defining (functions on) trees)
- Helpers.v (defining some helper lemmas that come in handy in the other files)
- FoldStep.v (trying to define $fold\_rightstep[]$ as a single function)
- Function.v (trying to define *step* using the *Function* keyword)

Introduction

The functional Pearl "On Building Trees of Minimum Height" by Richard S. Bird solves the problem of building a tree out of a list of sub-trees, in such a way that the resulting tree has the input list as its frontier while being minimum in the sense that any tree that we can build with this property has at least the same height. Where the height of a tree is defined as the maximum depth of any node node in the tree.

Local minimum pairs

When we are joining pairs together repeatedly, we always end up with a tree that has the input list as its frontier. This paper introduces the concept of 'local minimum pairs' (LMP), for which it is always 'safe' to join them, in respect to the height of the resulting tree.

Lemma 1

The author proves a lemma (which we will call 'Lemma 1') that says that we can safely join a local minimum pair in the process of building a tree. Where safely means that we do not lose the opportunity of constructing a tree that has a minimum height. In other words, to find a tree of minimum height we can always select the first lmp that we encounter in our input list and join it. Since any list has always at least one lmp this process will halt in a single tree that is of minimum height.

The main definition

The author provides us withan algorithm that consists of two steps. One 'pre-processing' step that will produce a list of trees in such a way that this list is strictly increasing (using the height of the trees as cost function), and a final step that will transform this preprocessed list (which might already contain but a single element) into a single tree. Since we show that for any strictly increasing list, the left-most pair is always an lmp, using $foldl\ join$ on this list will create a tree of minimal height.

build = foldl1 join . fold_right step []

This algorithm works in linear time due to the preprocessing. In order to prove the correctness of this algorithm we will split up the work in two phases (each on one end of the function composition operator): * First that $fold\_right\ step$ [] produces a strictly increasing list * Second that $foldl1\ join$, given a strictly increasing list, produces a tree of minimum height.

Proving the tree is of minimum height

Proving that $foldl1\ join$ produces a tree of minimum height, relies strongly on the so called 'Lemma 1'. We can take two approaches:

- Proving Lemma 1 is correct and that $foldl1\ join$ always selects an lmp, when given a strictly increasing list

- Proving that *foldl*1 *join* produces a tree of minimal height, when given a strictly increasing list

Both approaches will be equally correct, however in the first approach the link between Lemma 1 and *foldl*1 *join* is not explicit (only implied). This, however, makes it easier to prove, due to separation of concerns.

We assume that the definition of *fold_left* that we have imported is indeed

correct. The loop *foldl*1 *join* will always use join, unless the input list contains but a single element. From the definition of *fold_left* we know that *foldl*1 *join* will always join the first pair of a list.

All we have to show is that every join operation performed by *fold_left* is 'safe' (invariant). Where a safe join means that the joined pair is a local minimum pair in the given input list. Which comes down to showing that the left-most pair in a strictly increasing list is always an lmp (precondition) and that appending the join of the left-most pair of a strictly increasing list preserves the fact that the left-most pair of this list is an lmp even though a list produced this way does not necessarily have to be strictly increasing anymore (invariant is preserved).

The proofs can be found in the file Minimum.v.v and are named *s_inc_leftmost_lmp* and *join_preserves_leftmost_lmp*. Lemma 1 can also be found at this location.

Definition of foldl1

We have defined foldl1 (which is usually not total) using a predicate that the input list is not empty, which allows us to ignore the usual base case of foldl.

Proving strictly increasingness

In order to prove that *fold_right step* [] produces a strictly increasing list we will have to do case analysis on the function step.

Function step acts very much like the list constructor *cons*, accept that it analyzes the first elements of the list that it will add to by pattern matching. When the newly element is smaller then the element currently on top of the list we can just add it, otherwise we will join the new element and the head of the list together and add this joined tree to the list instead (again using step). If we try to use *step* as a compositional operator for creating a decreasing list (by giving it values in decreasing order), the result will simply be a single tree since join will be applied to all elements.

Using *join* is only safe (looking at the properties that the entire algorithm should have) when the joined pair is an lmp. This is not, however, the reason that the author chose to examine the first two elements of the list, as we will discuss later.

In order to define the function step, which uses nested recursion and is therefor not clearly structural decreasing, we have tried several methods used for defining 'general recursive functions'. Including

- Using the keyword *Function*
- Define *fold_right step* [] as a single function, since *fold_right* has a an obvious decreasing argument.
- Using the Bove-Capretta method.
- Using Well-foundedness
- *Using an syntactical alternative definition, which is semantically equivalent*
- Using a decreasing natural number and ensuring that it is always at least as big as the length of the input list of which we know that is decreases. We pattern match on this natural number and give a bogus result when it is zero. Meaning that if we want to prove that the function step as certain properties it can not return this bogus result (the result is only bogus when the length of the worklist is zero itself). This corresponds very much to an amortizing argument, giving the function step a limit on the number of recursive calls it can use.

Keyword Function

Using the keyword *Function* seemed like a very good approach since we expected to be able to {*measure length input*}, where *input* is the input list. However, *Function* can not be used to define functions with nested recursion (a recursive call of which the result is an argument to a recursive call). (The attempt can be found in Function.v)

Bove-Capretta

The same problem we have encountered when using the Bove-Capretta method. Similar to Bove and Capretta's example of QuickSort in their paper on 'General Recursion', they provided an example of a function that has a nested recursive call. Bove and Capretta described in their paper a problem about nested recursive calls, namely that the predicate and function we want to define depend on each other mutually. This is exactly the problem we encountered while defining the function *step*. The provided solution in the paper is based on work of Dybjer (2000), who introduced a method of defining a predicate and the definition it supports at the same time, even though they depend on each other. This method does, however, not help us to define the function / predicate pair in Coq.

Defining FoldStep

Another unsuccessful approach was trying to define *fold_right step* [] as a single function (see FoldStep.v). The function is defined as to have two list of trees as arguments. One being the 'worklist', while the other is the 'accumulated result'. When the worklist is empty we simply return the accumulated result, otherwise we add the first element to the worklist to the accumulated result in such a way that it should match to definition of step in the paper by Bird. This new accumulated result is then given as an argument to a recursive call of the function *fold_step* together with the decreased worklist. When we reach the cases of the function step where we need a nested recursive call, due to the nature of the definition of *fold_step*, we have to invent a new worklist which contains only a single element. This seems to be the problem of this definition, since we have no certainty that the worklist of the current function call is actually larger then a single element. Which means that we can not be sure that the old worklist is always larger then the new worklist, since the old and the new worklist can both be of length one.

Amortizing argument

A method that we did use successfully, was to use something very similar to an amortizing argument. Namely we give the function step a natural number as additional argument and make sure that this number is always decreased when being passed on with every recursive call. In order to assure that this can happen we need to pattern match on it and provide a function result when the natural number reaches zero. The result we give in this case is arbitrary and to prevent this result from ever being returned we must give the initial call a value high enough natural number to begin with. This number has to be related to the length of the input list and can not be arbitrary since we always increase the size of the input list to such an extent that any arbitrary number is not enough. However, we can see, by analyzing the algorithm (the paper suggests an amortizing approach), that the input list decreases at least by one in every recursive call. And since our natural number will exactly decrease by one every recursive call, we can use the length of the input list as the initial value of this extra argument. We were successful in defining the function step using this method, although unable to prove the properties that we wanted, as will be discussed later.

Well-founded recursion

Studying the chapter on 'Well-founded recursion' by Adam Chlipala we came to the conclusion that we might well run into the same problem we faced when attempting the Bove-Capretta method. The function that 'splits' the recursive argument is namely the function we are trying to define itself, unlike *filter* that is being used for defining QuickSort in the Bove-Capretta example and unlike *split* that is being used in the MergeSort example in the chapter on 'Well-founded recursion'.

Alternative Definition

Thinking about how to show that our nested recursive call gives back a list which is smaller then the input case we did came up with an alternative definition of the function step, which is semantically the same. Looking back at the original definition of the function *step* in the paper there is only one problem to solve, namely the case

t >= u /\ t >= v

since it is not obvious that

step (join u v) ts

is structurally smaller then $u :: v :: ts$. However, Coq also complains about the case

t >= u /\ t < v

since the recursive call should have an argument ts as opposed to $v :: ts$. We solved this issue, by separating the case analysis of the input list into

```
match xs with
| nil => ...
| u :: vs =>
    match vs with
    | nil => ...
    | v :: ts => ...
```

which allows us to give vs as an argument to *step* (*join t u*) instead of $(v :: ts)$, which is accepted by Coq since it is not a function application. However, the real problem, as mentioned before, is defining

```
    step t (step (join u v) ts)
```

where

```
    step (join u v) ts
```

should be structurally smaller then $u :: v :: ts$. We solved this problem by replacing this recursive call by a call to another function which behaves semantically the same.

Lets take a look at how this function should behave. As mentioned before, the function step acts exactly like the *cons* constructor of a list, except that it will *join* the new element to be inserted with the head of the list when the new elements height is not smaller the that of the head of the list. This behavior corresponds directly to the definition of the function called *join_until_smaller* which can be found in SInc.v

Proofs about the results of $fold\ step$ []

We have to show that $fold\ step$ [] produces a non-empty list (in order to use $foldl1$) and have to show that this list is strictly increasing (to guarantee that $foldl1\ join$ will always join local minimum pairs). The proofs can be found in SInc.v and their theorems are named *fold_step_not_nil* and *fold_step_inc*, respectively. Both proofs rely heavily on the fact that these properties hold for the function *step* itself. Which is shown in the proofs of theorems *step_inc* and *step_not_nil*.

We were able to give these proofs for our alternative definition of step (see SInc.v), but not for our definition using an amortizing argument (see StepN.v). The difference lies in the fact that the alternative definition of step no longer has a nested recursive call, which requires just the right induction hypothesis, and is therefor much easier.

Open endings

Given the theorems and definitions we have shown that the algorithm proposed in the paper ‘On building trees of minimum height’ by Richard S. Bird is indeed correct, meaning that the algorithm produces a tree of minimum from an input list of subtrees and doing so in such a way that the frontier of the result tree is the same as the input list. However, there are some open endings to our story. Namely:

- We have not proved the correctness of Lemma 1, which can be considered a big miss since the correctness of the algorithm relies on it. However, Bird has proven it to be correct in the paper itself, which made us decide to focus on (correctness of) the definitions used in the paper.

- Prove $foldl1\ join \Rightarrow minimum$ using Lemma 1 and a definition of *minimum* explicitly, in such a way that it can be used in the proof for function *build*. We have only shown this implicitly, meaning that we did not relate $foldl1join$ directly to Lemma 1 or the definition of *minimum*.
- Proving explicitly that join is only applied to local minimum pairs.

Proving explicitly that join is only applied to local minimum pairs.

So far we have always assumed that join is only applied to local minimum pairs. Except for the joins used in $foldl1\ join$, we have shown that they are always lmps. Relying on the fact that in every case a *join* is used, it should be clear which of the lmps constructors apply.

To be more correct we could have made this explicit by changing the definition of *join* in such a way that it only works on pairs of which a proof that they are a local minimum pair can be given. However, two trees are only a local minimum pair in respect to their context, meaning that we have to change *join* not only to receive two trees as input but also the list in which the two trees are a member.

Not only will this definitions make all the other definitions (and proofs) much more cluttered, we can also fool this definition by always giving it the list with only the two elements of the pair and using the *s_inc_two* constructor to make the proof that they are a local minimum pair (for the list in which only they are present).