Verifying Richard Bird’s “On building trees of minimum height”

L.T. van Binsbergen  J.P. Pizani Flor

Department of Information and Computing Sciences, Utrecht University

Wednesday 26th June, 2013
“Combining a list of trees”

Given a list of trees, build a tree (of minimum height) that has the elements of the list as frontier (preserving order).

- We want to minimize cost, where cost means:

  \[
  \text{cost } t = \left( \max_{1 \leq i \leq N} \text{depth}_i + h_i \right)
  \]

- depth\(_i\) is the length of a path from root to tip \(i\)
- \(h_i\) is the height of the \(i^{th}\) element of the input list
Simpler but equivalent problem

The problem can be stated with natural numbers instead of trees being the elements of the input list.

- \( hs = [h_1, h_2, \ldots, h_N] \)
- Each element of the list is then considered the \textit{height} of the tree.
- We use this “simplified” form of the problem in an example, but the “full” form is the one verified.
The basis of the algorithm proposed is the concept of a “local minimum pair”:

- A pair \((t_i, t_{i+1})\) in a sequence \(t_i(1 \leq i \leq N)\) with heights \(h_i\) such that:
  - \(\max(h_{i-1}, h_i) \geq \max(h_i, h_{i+1}) < \max(h_{i+1}, h_{i+2})\)

- An alternative set of conditions, used in the proof of correctness:
  - \(h_{i+1} \leq h_i < h_{i+2}\), or
  - \((h_i < h_{i+1} < h_{i+2}) \land (h_{i-1} \geq h_{i+1})\)
Greedy algorithm - example

- There is \textit{at least} one LMP, the rightmost one.
- The algorithm combines the rightmost LMP at each stage.
- Example in the whiteboard...
Correctness of the algorithm

The correctness of this algorithm relies fundamentally on the so-called “Lemma 1”:

“Suppose that \((t_i, t_{i+1})\) in an lmp in a given sequence of trees \(t_j(1 \leq j \leq N)\). Then the sequence can be combined into a tree \(T\) of minimum height in which \((t_i, t_{i+1})\) are siblings.”
Correctness of the algorithm

The correctness of this algorithm relies fundamentally on the so-called “Lemma 1”:

“Suppose that \((t_i, t_{i+1})\) in an \(lmp\) in a given sequence of trees \(t_j(1 \leq j \leq N)\). Then the sequence can be combined into a tree \(T\) of \textbf{minimum height} in which \((t_i, t_{i+1})\) are \textbf{siblings}.”

- In the paper, the proof of this lemma is done \textit{by contradiction} and case analysis on whether the trees are \textit{critical}.
Correctness of the algorithm

How we expressed “Lemma 1” in Coq:

Theorem Lemma1: forall (l s : list tree) (a b : tree) (sub : l = [a;b] ++ s), lmp a b l -> exists (t : tree), siblings t a b -> minimum l t.
Proof.
Admitted.

Fixpoint siblings (t : tree) (a b : tree) : Prop := match t with |
  Tip _ => False |
  Bin _ x y => a = x \/ b = y \/ siblings x a b \/ siblings y a b end.

Definition minimum (l : list tree) (t : tree) : Prop := forall (t’ : tree), flatten t’ = l -> ht t <= ht t’.
The “build” function and \textit{foldl1}

The “top level” function of the algorithm looks like this:

\begin{verbatim}
build = foldl1 join . foldr step []
\end{verbatim}

- The first big issue we face is how to describe a \textbf{total} version of \textit{foldl1} in Coq.
The “build” function and \textit{foldl1}

The “top level” function of the algorithm looks like this:

\begin{verbatim}
build = foldl1 join . foldr step []
\end{verbatim}

- The first big issue we face is how to describe a \textbf{total} version of \textit{foldl1} in Coq.
- We modeled this by passing a proof that the list is non-empty:

\textbf{Definition} \textit{foldl1} \((f : \text{tree} \to \text{tree} \to \text{tree}) \ (l : \text{list} \text{tree}) \ (P : l <> \text{nil}) : \text{tree}.

\text{case} \ l \ \text{as} \ \li \ x \ \text{xs}].

\text{contradiction} \ P.

\text{reflexivity}.

\text{apply} \ \text{fold\_left} \ \text{with} \ (B := \text{tree}).

\text{exact} \ f. \ \text{exact} \ \text{xs}. \ \text{exact} \ x.

\text{Defined}.

Non-structural recursion in \textit{step}

The other BIG issue faced by us is the use of non-structural recursion in the function \textit{step}:

\begin{verbatim}
step t [] = [t]
step t [u]
    | ht t < ht u = [t,u]
    | otherwise   = [join t u]
step t (u : v : ts)
    | ht t < ht u = t : u : v : ts
    | ht t < ht v = step (join t u) (v : ts)
    | otherwise   = step t (step (join u v) ts)
\end{verbatim}

We tried:

- “Function” keyword.
- \textit{Bove-Capretta}
  - Termination predicate and \textit{step} are \textit{mutually recursive}.
- Define \textit{step} using structural recursion on a natural \( n \geq len(l) \).