PMT Additional Exercises (Week 5)

Nicholas Sim

February 8th, 2018

1 Maximum sum

```haskell
maxSum :: [Integer] -> Integer
maxSum = snd . foldr f (0,0)
  where f x (c,g) = (c',g')
    where c' = max 0 (c+x)
          g' = max g c'
```

Prove that `maxSum` (Kadane) computes the maximum sum of a *contiguous* subarray. That is:

$$\forall \texttt{xs:[Int].maxSum xs} = \max_{0 \le i \le j < \#\texttt{xs}} \sum_{k=i}^{j} \texttt{xs}[k]$$

Remark: this algorithm is usually run left-to-right (i.e. `foldl`), but produces the same result.

2 Cycle-finding

Remark: representing (potentially) cyclic data structures in Haskell can be a hazard. Search 'tying the knot' for more information.

```haskell
data Node a = Node { value :: a, targets :: [Node a] }

findCycle :: [Node a] -> Node a -> Bool
findCycle visited node
= elem node visited || any (findCycle node:visited) (targets x)
```

A `Node` represents a node in a connected *directed* graph. Use $cycle(\texttt{n})$ for a predicate indicating a cycle can be reached from a node `n`, and $reach(\texttt{n}, \texttt{m})$ for `n` is reachable from `m` (both nodes).

1. Write down a structural induction principle for the `Node` data type.
2. (*) Write down a property (provable with induction) sufficient to show that `findCycle` determines if there is a cycle from a given node.
3. (***) Complete the proof[1]

[1]Unlike last week, I have a written solution. You should consider this question to be at least as difficult as any exam question for Reasoning.

3 Solution to maximum sum

This is a classic problem in computer science, and as noted, we have implemented Kadane's algorithm, which has linear time complexity.

The first thing we should observe is that `c` represents the current sum, and `g` the global. Similar to last week, we will instead prove a property about `f`:

$$\forall \texttt{xs:[Int].foldr f (0,0) xs} = (\max_{0 \le j < \#\texttt{xs}} \sum_{k=0}^{j} \texttt{xs}[k], \max_{0 \le i \le j < \#\texttt{xs}} \sum_{k=i}^{j} \texttt{xs}[k])$$

Recall:

```
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Base case To show: `foldr f (0,0) [] = (0,0)`

This follows directly from the definition of `foldr`.

Inductive step Fix `xs:[Int]`.

Inductive Hypothesis:

$$\texttt{foldr f (0,0) xs} = (\max_{0 \le j < \#\texttt{xs}} \sum_{k=0}^{j} \texttt{xs}[k], \max_{0 \le i \le j < \#\texttt{xs}} \sum_{k=i}^{j} \texttt{xs}[k])$$

We will call this tuple `(c,g)`.

To show:

$$\forall \texttt{x:Int.foldr f (0,0) x:xs} = (\max_{0 \le j \le \#\texttt{xs}} \sum_{k=0}^{j} \texttt{(x:xs)}[k], \max_{0 \le i \le j \le \#\texttt{xs}} \sum_{k=i}^{j} \texttt{(x:xs)}[k])$$

Claim (*) that `(c,g)` represent the variables mentioned above at each step of the fold. Alternate I.H.:

$$\begin{array}{ll}
\texttt{maxSum x:xs} & \\
= \texttt{snd . foldr f (0,0) x:xs} & \text{by def. maxSum} \\
= \texttt{snd . f x (foldr f (0,0) xs)} & \text{by def. foldr} \\
= \texttt{snd . f x (c,g)} & \text{by I.H.} \\
= \texttt{snd . (max 0 c+x,max g (max 0 c+x))} & \text{by def. f} \\
= \texttt{max g (max 0 c+x)} & \text{by def. snd}
\end{array}$$

I claim that this is what we want. Cases: either `x` is in the sum, or not. If it is, the sublist starts at `x`, thus `c+x` is the maximum. Otherwise `g` is the maximum (by problem definition and given by I.H.).

4 Solution to cycle-finding

Remark: this is for reachable nodes. We only have one case (refreshing!)[2]

$$\forall \texttt{n:Node}.(\forall \texttt{m} \in \texttt{targets n}.P(\texttt{m})) \to P(\texttt{n})$$

Want to show: $\forall$`n:Node,vs:[Node].findCycle vs n` $\leftrightarrow cycle(\texttt{n}) \lor \exists \texttt{v} \in \texttt{vs} : reach(\texttt{v},\texttt{n})$[3]

This proof is a sketch only, write the detail in yourself.

[2]Below, we use $\forall \texttt{x} \in \texttt{xs}.P(\texttt{x})$ as a shorthand for $\forall \texttt{x:a.elem x xs} \to P(\texttt{x})$, where `xs:[a]` and P some predicate.

[3]Likewise, $\exists \texttt{x} \in \texttt{xs}.P(\texttt{x})$ represents *something sensible that I am too lazy to write.* (sufficient to apply double negative to the universally quantified case to note that this is valid)

Inductive step Fix `ts:[Node]`.

Inductive Hypothesis:

$$\forall \mathtt{m} \in \mathtt{ts}, \mathtt{vs:[Node]}.\mathtt{findCycle\ vs\ m} \leftrightarrow cycle(\mathtt{m}) \vee \exists \mathtt{v} \in \mathtt{vs} : reach(\mathtt{v}, \mathtt{m})$$

To show:

$$\forall \mathtt{x:a}, \mathtt{vs:[Node]}.\mathtt{findCycle\ vs\ n} \leftrightarrow cycle(\mathtt{n}) \vee \exists \mathtt{v} \in \mathtt{vs} : reach(\mathtt{v}, \mathtt{n})$$

writing `n` for `Node {value=x, targets=ts}`.

Fix `x:a`, `vs:[Node]`.

Assume LHS. Want to show $cycle(\mathtt{n}) \vee \exists \mathtt{v} \in \mathtt{vs} : reach(\mathtt{v}, \mathtt{n})$. Assume $\mathtt{n} \notin \mathtt{vs}$, otherwise $reach$ immediately follows from `elem node visited`. So $\exists \mathtt{m} \in \mathtt{ts}$: `findCycle n:vs m`. Applying I.H. we get $cycle(\mathtt{m})$ (so $cycle(\mathtt{n})$) or $\exists \mathtt{v} \in (\mathtt{n:vs}) : reach(\mathtt{v}, \mathtt{m})$ (in which case $\exists \mathtt{v} \in \mathtt{vs} : reach(\mathtt{v}, \mathtt{n})$, or $reach(\mathtt{n}, \mathtt{n})$, i.e. $cycle(\mathtt{n})$).

Assume $\exists \mathtt{v} \in \mathtt{vs} : reach(\mathtt{v}, \mathtt{n})$. Then certainly either $\mathtt{n} \in \mathtt{vs}$ (hence `elem n vs`), or $\exists \mathtt{m} \in \mathtt{ts} : \exists \mathtt{v} \in \mathtt{vs} : reach(\mathtt{v}, \mathtt{m})$ (fulfil `any` case using I.H.). Thus LHS holds.

Finally, for the substance of the proof: assume $cycle(\mathtt{n})$. Clearly we have $reach(\mathtt{n}, \mathtt{n})$ (`n` is part of a cycle), or $\exists \mathtt{m} \in \mathtt{ts} : cycle(\mathtt{m})$. If the latter holds, then $\forall \mathtt{vs:[Node]}.\exists \mathtt{m} \in \mathtt{ts}$: `findCycle vs m` by I.H., so the `any` case holds. Otherwise, `n` is in `ts` (trivial application of the `elem` case), or $\exists \mathtt{m} \in \mathtt{ts} : reach(\mathtt{n}, \mathtt{m})$ (by reasoning about cycles and reachability). But now `findCycle n:vs m` holds since $\exists v(n : vs) : reach(\mathtt{v}, \mathtt{m})$ (I.H.), so `findCycle vs n` holds by the `any` case.