Programming Languages: Functional Programming Practicals 5. Program Calculation

Shin-Cheng Mu

Spring, 2020

1. Consider the internally labelled binary tree:

$$\textbf{data}\ \mathsf{ITree}\ a = \mathsf{Null} \mid \mathsf{Node}\ a\ (\mathsf{ITree}\ a)\ (\mathsf{ITree}\ a)\ \ .$$

   (a) Define $sumT :: \mathsf{ITree\ Int} \rightarrow \mathsf{Int}$ that computes the sum of labels in an ITree.

   (b) A *baobab tree* is a kind of tree with very thick trunks. An Itree Int is called a baobab tree if every label in the tree is larger than the sum of the labels in its two subtrees. The following function determines whether a tree is a baobab tree:

$$\begin{array}{l}
baobab :: \mathsf{ITree\ Int} \rightarrow \mathsf{Bool} \\
baobab\ \mathsf{Null} \qquad\qquad = \mathsf{True} \\
baobab\ (\mathsf{Node}\ x\ t\ u) = baobab\ t \wedge baobab\ u \wedge \\
\qquad\qquad\qquad\qquad\quad x > (sumT\ t + sumT\ u)\ \ .
\end{array}$$

   What is the time complexity of $baobab$? Define a variation of $baobab$ that runs in time proportional to the size of the input tree by tupling.

2. Recall the externally labelled binary tree:

$$\textbf{data}\ \mathsf{Etree}\ a = \mathsf{Tip}\ a \mid \mathsf{Bin}\ (\mathsf{ETree}\ a)\ (\mathsf{ETree}\ a)\ \ .$$

   The function $size$ computes the size (number of labels) of a tree, while $repl\ t\ xs$ tries to relabel the tips of t using elements in xs. Note the use of $take$ and $drop$ in $repl$:

$$\begin{array}{l}
size\ (\mathsf{Tip}\ _) \quad\ = 1 \\
size\ (\mathsf{Bin}\ t\ u) = size\ t + size\ u\ \ . \\
repl :: \mathsf{ETree}\ a \rightarrow \mathsf{List}\ b \rightarrow \mathsf{ETree}\ b \\
repl\ (\mathsf{Tip}\ _)\quad xs = \mathsf{Tip}\ (head\ xs) \\
repl\ (\mathsf{Bin}\ t\ u)\ xs = \mathsf{Bin}\ (repl\ t\ (take\ n\ xs))\ (repl\ u\ (drop\ n\ xs)) \\
\quad \textbf{where}\ n = size\ t\ \ .
\end{array}$$

   The function $repl$ runs in time $O(n^2)$ where n is the size of the input tree. Can we do better? Try discovering a linear-time algorithm that computes $repl$. **Hint**: try calculating the following function:

$$
\begin{aligned}
&\mathit{repTail} :: \mathsf{ETree}\ a \to \mathsf{List}\ b \to (\mathsf{ETree}\ b, \mathsf{List}\ b)\\
&\mathit{repTail}\ s\ \mathit{xs} = (???, ???)\ \ ,\\
&\quad \mathbf{where}\ n = \mathit{size}\ s\ \ ,
\end{aligned}
$$

where the function $\mathit{repTail}$ returns a tree labelled by some prefix of xs, together with the suffix of xs that is not yet used (how to specify that formally?).

You might need properties including:

$$
\begin{aligned}
\mathit{take}\ m\ (\mathit{take}\ (m+n)\ \mathit{xs}) &= \mathit{take}\ m\ \mathit{xs}\ \ ,\\
\mathit{drop}\ m\ (\mathit{take}\ (m+n)\ \mathit{xs}) &= \mathit{take}\ n\ (\mathit{drop}\ m\ \mathit{xs})\ \ ,\\
\mathit{drop}\ (m+n)\ \mathit{xs} &= \mathit{drop}\ n\ (\mathit{drop}\ m\ \mathit{xs})\ \ .
\end{aligned}
$$

3. The function tags returns all labels of an internally labelled binary tree:

$$
\begin{aligned}
&\mathit{tags} :: \mathsf{ITree}\ a \to \mathsf{List}\ a\\
&\mathit{tags}\ \mathsf{Null} = [\,]\\
&\mathit{tags}\ (\mathsf{Node}\ x\ t\ u) = \mathit{tags}\ t +\!\!+ [x] +\!\!+ \mathit{tags}\ u\ \ .
\end{aligned}
$$

Try deriving a faster version of tags by calculating

$$
\begin{aligned}
&\mathit{tagsAcc} :: \mathsf{ITree}\ a \to \mathsf{List}\ a \to \mathsf{List}\ a\\
&\mathit{tagsAcc}\ t\ \mathit{ys} = \mathit{tags}\ t +\!\!+ \mathit{ys}\ \ .
\end{aligned}
$$

4. Recall the standard definition of factorial:

$$
\begin{aligned}
&\mathit{fact} :: \mathsf{Nat} \to \mathsf{Nat}\\
&\mathit{fact}\ 0 = 1\\
&\mathit{fact}\ (\mathbf{1}_+\ n) = \mathbf{1}_+\ n \times \mathit{fact}\ n\ \ .
\end{aligned}
$$

This program implicitly uses space linear to n in the call stack.

   1. Introduce $\mathit{factAcc}\ n\ m = \ldots$ where m is an accumulating parameter.
   2. Express fact in terms of $\mathit{factAcc}$.
   3. Construct a space efficient implementation of $\mathit{factAcc}$.

5. Define the following function expAcc:

$$
\begin{aligned}
&\mathit{expAcc} :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}\\
&\mathit{expAcc}\ b\ n\ x = x \times \mathit{exp}\ b\ n\ \ .
\end{aligned}
$$

   (a) Calculate a definition of expAcc that uses only $O(\log n)$ multiplications to compute b^n. You may assume all the usual arithmetic properties about exponentials. **Hint**: consider the cases when n is zero, non-zero even, and odd.

(b) The derived implementation of $expAcc$ shall be tail-recursive. What imperative loop does it correspond to?

6. Recall the standard definition of Fibonacci:

$$
\begin{aligned}
&\mathit{fib} :: \mathsf{Nat} \rightarrow \mathsf{Nat} \\
&\mathit{fib}\ 0 &&= 0 \\
&\mathit{fib}\ 1 &&= 1 \\
&\mathit{fib}\ (\mathbf{1}_+\ (\mathbf{1}_+\ n)) &&= \mathit{fib}\ (\mathbf{1}_+\ n) + \mathit{fib}\ n \quad .
\end{aligned}
$$

Let us try to derive a linear-time, tail-recursive algorithm computing fib.

1. Given the definition $\mathit{ffib}\ n\ x\ y = \mathit{fib}\ n \times x + \mathit{fib}\ (\mathbf{1}_+\ n) \times y$, Express fib using ffib.
2. Derive a linear-time version of ffib.