Programming Languages: Functional Programming Practicals 5. Program Calculation

Shin-Cheng Mu

Spring, 2020

1. Consider the internally labelled binary tree:

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

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

Solution:

$$\begin{aligned}
&sumT :: \mathsf{ITree\ Int} \to \mathsf{Int}\\
&sumT\ \mathsf{Null} = 0\\
&sumT\ (\mathsf{Node}\ x\ t\ u) = x + sumT\ t + sumT\ u \quad .
\end{aligned}$$

(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:

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

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.

Solution: Define:

$$\begin{aligned}
&baosum :: \mathsf{Tree\ Int} \to (\mathsf{Bool}, \mathsf{Int})\\
&baosum\ t = (baobab\ t, sumT\ t) \quad .
\end{aligned}$$

such that $baobab = fst \cdot baosum$.

With $t{:=}\mathsf{Null}$, it is immediate that $baosum\ \mathsf{Null} = (\mathsf{True}, 0)$. Consider $t{:=}\mathsf{Node}\ x\ t\ u$:

$$
\begin{aligned}
& \quad \mathit{baosum}\ (\mathsf{Node}\ x\ t\ u) \\
= & \quad \{ \text{ definition of } \mathit{baosum} \ \} \\
& \quad (\mathit{baobab}\ (\mathsf{Node}\ x\ t\ u), \mathit{sumT}\ (\mathsf{Node}\ x\ t\ u)) \\
= & \quad \{ \text{ definitions of } \mathit{baobab} \text{ and } \mathit{sumT} \ \} \\
& \quad (\mathit{baobab}\ t \wedge \mathit{baobab}\ u \wedge x > (\mathit{sumT}\ t + \mathit{sumT}\ u), \\
& \quad\ x + \mathit{sumT}\ t + \mathit{sumT}\ u) \\
= & \quad \{ \text{ introducing local variables } \} \\
& \quad \mathbf{let}\ (b, y) = (\mathit{baobab}\ t, \mathit{sumT}\ t) \\
& \quad\quad\ \ (c, z) = (\mathit{baobab}\ u, \mathit{sumT}\ u) \\
& \quad \mathbf{in}\ (b \wedge c \wedge x > (y + z), x + y + z) \\
= & \quad \{ \text{ definition of } \mathit{baosum} \ \} \\
& \quad \mathbf{let}\ (b, y) = \mathit{baosum}\ t \\
& \quad\quad\ \ (c, z) = \mathit{baosum}\ u \\
& \quad \mathbf{in}\ (b \wedge c \wedge x > (y + z), x + y + z) \quad .
\end{aligned}
$$

We have thus derived:

$$
\begin{aligned}
& \mathit{baosum}\ \mathsf{Null} \qquad\qquad = (\mathsf{True}, 0) \\
& \mathit{baosum}\ (\mathsf{Node}\ x\ t\ u) = \\
& \quad \mathbf{let}\ (b, y) = \mathit{baosum}\ t \\
& \quad\quad\ \ (c, z) = \mathit{baosum}\ u \\
& \quad \mathbf{in}\ (b \wedge c \wedge x > (y + z), x + y + z) \quad .
\end{aligned}
$$

2. Recall the externally labelled binary tree:

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

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

$$
\begin{aligned}
& \mathit{size}\ (\mathsf{Tip}\ \_) \quad\ = 1 \\
& \mathit{size}\ (\mathsf{Bin}\ t\ u)\ = \mathit{size}\ t + \mathit{size}\ u \quad .
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{repl} :: \mathsf{ETree}\ a \to \mathsf{List}\ b \to \mathsf{ETree}\ b \\
& \mathit{repl}\ (\mathsf{Tip}\ \_) \quad \mathit{xs} = \mathsf{Tip}\ (\mathit{head}\ \mathit{xs}) \\
& \mathit{repl}\ (\mathsf{Bin}\ t\ u)\ \mathit{xs} = \mathsf{Bin}\ (\mathit{repl}\ t\ (\mathit{take}\ n\ \mathit{xs}))\ (\mathit{repl}\ u\ (\mathit{drop}\ n\ \mathit{xs})) \\
& \quad \mathbf{where}\ n = \mathit{size}\ t \quad .
\end{aligned}
$$

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:

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

where the function $repTail$ 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:

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

Solution: Define:

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

The case when$s := \mathsf{Tip}\ y$ is easy. Consider $s := \mathsf{Bin}\ t\ u$ (let $n1 = size\ t$, $n2 = size\ u$, and thus $size\ (\mathsf{Bin}\ t\ u) = n1 + n2$):

$$
\begin{aligned}
&\quad repTail\ (\mathsf{Bin}\ t\ u)\ xs \\
&= \quad \{\ \text{definition of } repTail\ \} \\
&\quad (repl\ (\mathsf{Bin}\ t\ u)\ (take\ (n1+n2)\ xs), drop\ (n1+n2)\ xs) \\
&= \quad \{\ \text{definition of } repl, \text{let } n1 = size\ t\ \} \\
&\quad (\mathsf{Bin}\ (repl\ t\ (take\ n1\ (take\ (n1+n2)\ xs))) \\
&\qquad\quad (repl\ u\ (drop\ n1\ (take\ (n1+n2)\ xs))), drop\ (n1+n2)\ xs) \\
&= \quad \{\ \text{property given}\ \} \\
&\quad (\mathsf{Bin}\ (repl\ t\ (take\ n1\ xs)) \\
&\qquad\quad (repl\ u\ (take\ n2\ (drop\ n1\ xs))), drop\ n2\ (drop\ n1\ xs)) \\
&= \quad \{\ \text{factoring common sub-expressions}\ \} \\
&\quad \mathbf{let}\ (t', xs')\ \ = (repl\ t\ (take\ n1\ xs), drop\ n1\ xs) \\
&\qquad\ (u', xs'') = (repl\ u\ (take\ n2\ xs'), drop\ n2\ xs') \\
&\quad \mathbf{in}\ (\mathsf{Bin}\ t'\ u', xs'') \\
&= \quad \{\ \text{definition of } repTail\ \} \\
&\quad \mathbf{let}\ (t', xs')\ \ = repTail\ t\ xs \\
&\qquad\ (u', xs'') = repTail\ u\ xs' \\
&\quad \mathbf{in}\ (\mathsf{Bin}\ t'\ u', xs'')\ \ .
\end{aligned}
$$

Thus we have:

$$
\begin{aligned}
&repTail\ (\mathsf{Tip}\ \_)\quad xs = (\mathsf{Tip}\ (head\ xs), tail\ xs) \\
&repTail\ (\mathsf{Bin}\ t\ u)\ xs = \mathbf{let}\ (t', xs')\ \ = repTail\ t\ xs \\
&\qquad\qquad\qquad\qquad\qquad\quad\ (u', xs'') = repTail\ u\ xs' \\
&\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ (\mathsf{Bin}\ t'\ u', xs'')\ \ .
\end{aligned}
$$

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

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

Try deriving a faster version of $tags$ by calculating

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

Solution: Apparently $tagsAcc\ \mathsf{Null}\ ys = ys$. Consider the case $t := \mathsf{Node}\ x\ t\ u$:

$$
\begin{aligned}
&\quad tagsAcc\ (\mathsf{Node}\ x\ t\ u)\ ys \\
&= tags\ (\mathsf{Node}\ x\ t\ u) +\!\!+ ys \\
&= (tags\ t +\!\!+ [x] +\!\!+ tags\ u) +\!\!+ ys \\
&= \quad \{ \text{ associativity of } (+\!\!+) \ \} \\
&\quad tags\ t +\!\!+ (x : tags\ u +\!\!+ ys) \\
&= tagsAcc\ t\ (x : tagsAcc\ u\ ys) \quad .
\end{aligned}
$$

We thus have

$$
\begin{aligned}
&tagsAcc\ \mathsf{Null} \qquad\qquad ys = ys \\
&tagsAcc\ (\mathsf{Node}\ x\ t\ u)\ ys = tagsAcc\ t\ (x : tagsAcc\ u\ ys) \quad .
\end{aligned}
$$

4. Recall the standard definition of factorial:

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

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

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

Solution: To exploit associativity of $(\times)$, we define:

$$\mathit{factAcc}\ n\ m = m \times \mathit{fact}\ n \quad .$$

We recover fact by letting

$$\mathit{fact}\ n = \mathit{factAcc}\ n\ 1 \quad .$$

To construct $\mathit{factAcc}$ we derive:
Case $n := 0$:

$$\begin{aligned}
& \mathit{factAcc}\ 0\ m \\
= & \quad \{ \text{ definition of } \mathit{factAcc} \} \\
& m \times \mathit{fact}\ 0 \\
= & \quad \{ \text{ definition of } \mathit{fact} \} \\
& m \quad .
\end{aligned}$$

Case $n := \mathbf{1}_+\ n$:

$$\begin{aligned}
& \mathit{factAcc}\ (\mathbf{1}_+\ n)\ m \\
= & \quad \{ \text{ definition of } \mathit{factAcc} \} \\
& m \times \mathit{fact}\ (\mathbf{1}_+\ n) \\
= & \quad \{ \text{ definition of } \mathit{fact} \} \\
& m \times ((\mathbf{1}_+\ n) \times \mathit{fact}\ n) \\
= & \quad \{ (\times) \text{ associative } \} \\
& (m \times (\mathbf{1}_+\ n)) \times \mathit{fact}\ n \\
= & \quad \{ \text{ definition of } \mathit{factAcc} \} \\
& \mathit{factAcc}\ n\ (m \times (\mathbf{1}_+\ n)) \quad .
\end{aligned}$$

Thus,

$$\begin{aligned}
\mathit{factAcc}\ 0\ & m = m \\
\mathit{factAcc}\ (\mathbf{1}_+\ n)\ & m = \mathit{factAcc}\ n\ (m \times (\mathbf{1}_+\ n)) \quad .
\end{aligned}$$

5. Define the following function expAcc:

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

   (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.

Solution: In the calculation below we write $exp\ b\ n$ as b^n, to be concise.

Apparently $expAcc\ b\ 0\ x = x$. For the case when n is even, that is $n := 2 \times n$:

$$\begin{aligned}
& expAcc\ b\ (2 \times n)\ x \\
=\ & x \times b^{2\times n} \\
=\ & \quad \{ \text{ since } b^{m\times n} = (b^m)^n \ \} \\
& x \times (b^2)^n \\
=\ & \quad \{ \text{ definition of } expAcc,\ b^2 = b \times b \ \} \\
& expAcc\ (b \times b)\ n\ x \quad .
\end{aligned}$$

For the case when n is odd, that is $n := 1 + n$:

$$\begin{aligned}
& expAcc\ b\ (1 + n)\ x \\
=\ & x \times b^{1+n} \\
=\ & \quad \{ \text{ definition of } exp \ \} \\
& x \times (b \times b^n) \\
=\ & \quad \{ \text{ associativity of } (\times) \ \} \\
& (x \times b) \times b^n \\
=\ & \quad \{ \text{ definition of } expAcc \ \} \\
& expAcc\ b\ n\ (x \times b) \quad .
\end{aligned}$$

We have derived:

$$\begin{aligned}
& expAcc\ b\ 0 && x = x \\
& expAcc\ b\ (2 \times n) && x = expAcc\ (b \times b)\ n\ x \\
& expAcc\ b\ (1 + n) && x = expAcc\ b\ n\ (x \times b) \quad .
\end{aligned}$$

In Haskell syntax, it is written:

$$\begin{aligned}
& expAcc\ b\ 0\ x && = x \\
& expAcc\ b\ n\ x \mid even\ n && = expAcc\ (b \times b)\ (n \text{ `}div\text{` } 2)\ x \\
& \qquad\qquad\quad \mid odd\ n && = expAcc\ b\ (n - 1)\ (x \times b) \quad .
\end{aligned}$$

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

Solution: To calculate B^{N}:

$$\begin{aligned}
& b, n, x := \mathsf{B}, \mathsf{N}, 1; \\
& \mathbf{do}\ n \neq 0 \rightarrow \mathbf{if}\ even\ n \rightarrow b, n := b \times b, n \text{ `}div\text{` } 2 \\
& \qquad\qquad\quad\ \mid\ odd\ \ n \rightarrow n, x := n - 1, x \times b \\
& \qquad\qquad\quad\ \mathbf{fi} \\
& \mathbf{od};
\end{aligned}$$

$$return\ x$$

The loop invariant is $\mathsf{B}^{\mathsf{N}} = x \times b^n$.

6. Recall the standard definition of Fibonacci:

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

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.

Solution: $\mathit{fib}\ n = \mathit{ffib}\ n\ 1\ 0$.

To construct ffib, we calculate:
Case $n := 0$:

$$\begin{aligned}
&\ \ \mathit{ffib}\ 0\ x\ y\\
=&\quad \{\ \text{definition of } \mathit{ffib}\ \}\\
&\ \ \mathit{fib}\ 0 \times x + \mathit{fib}\ 1 \times y\\
=&\quad \{\ \text{definition of } \mathit{fib}\ \}\\
&\ \ 0 \times x + 1 \times y\\
=&\ y\ \ .
\end{aligned}$$

Case $n := \mathbf{1}_+\ n$:

$$\begin{aligned}
&\ \ \mathit{ffib}\ (\mathbf{1}_+\ n)\ x\ y\\
=&\quad \{\ \text{definition of } \mathit{ffib}\ \}\\
&\ \ \mathit{fib}\ (\mathbf{1}_+\ n) \times x + \mathit{fib}\ (\mathbf{1}_+\ (\mathbf{1}_+\ n)) \times y\\
=&\quad \{\ \text{definition of } \mathit{fib}\ \}\\
&\ \ \mathit{fib}\ (\mathbf{1}_+\ n) \times x + (\mathit{fib}\ (\mathbf{1}_+\ n) + \mathit{fib}\ n) \times y\\
=&\quad \{\ \text{arithmetics}\ \}\\
&\ \ \mathit{fib}\ (\mathbf{1}_+\ n) \times (x + y) + \mathit{fib}\ n \times y\\
=&\quad \{\ \text{definition of } \mathit{ffib}\ \}\\
&\ \ \mathit{ffib}\ n\ y\ (x + y)\ \ .
\end{aligned}$$

Therefore,

$$\begin{aligned}
&\mathit{ffib}\ 0\ &&x\ y = y\\
&\mathit{ffib}\ (\mathbf{1}_+\ n)\ &&x\ y = \mathit{ffib}\ n\ y\ (x + y)\ \ .
\end{aligned}$$