Programming Languages: Functional Programming Practicals 5. Program Calculation

Shin-Cheng Mu

Autumn, 2023

1. Consider the internally labelled binary tree:

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

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

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

   (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:

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

   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:
>
> $$\begin{aligned} &baosum :: \mathsf{Tree\ Int} \to (\mathsf{Bool}, \mathsf{Int}) \\ &baosum\ t = (baobab\ t, sumT\ t)\ . \end{aligned}$$
>
> such that $baobab = fst \cdot baosum$.
> With t:=Null, it is immediate that $baosum\ \mathsf{Null} = (\mathsf{True}, 0)$. Consider t:=Node $x\ t\ u$:

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

We have thus derived:

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

2. Recall the externally labelled binary tree:

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

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 $repl$:

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

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:

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

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:

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

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

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

Thus we have:

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

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

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

Try deriving a faster version of $tags$ by calculating

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

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

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

We thus have

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

4. Recall the standard definition of factorial:

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

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:

$$factAcc\ n\ m = m \times fact\ n\ \ .$$

We recover $fact$ by letting

$$fact\ n = factAcc\ n\ 1\ \ .$$

To construct $factAcc$ we derive:
Case $n := 0$:

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

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

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

Thus,

$$\begin{array}{l} factAcc\ 0 \qquad m = m \\ factAcc\ (\mathbf{1}_+\ n)\ m = factAcc\ n\ (m \times (\mathbf{1}_+\ n))\ \ . \end{array}$$

5. Define the following function $expAcc$:

$$\begin{array}{l} expAcc :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\ expAcc\ b\ n\ x = x \times exp\ b\ n\ \ . \end{array}$$

   (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 $n := 2 \times n$:

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

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

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

We have derived:

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

In Haskell syntax, it is written:

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

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

Solution: To calculate B^{N}:

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

$return\ x$

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 $n := 0$:

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

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

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

Therefore,

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