2008 Formosan Summer School on Logic, Language, and Computation

Program Construction and Reasoning Exercises for Day 1

Shin-Cheng Mu

July 4th, 2008

1 In-Class Exercises

1.1 The Expand/Reduce Transformation

1. (a) What does this function do?

$$\begin{array}{lcl} \mathit{descend}\ 0 & = & [\,] \\ \mathit{descend}\ (n+1) & = & (n+1){:}\mathit{descend}\ n \end{array}$$

Ans:
$\mathit{descend}\ n = [n, n-1, (n-2), \cdots, 1]$

(b) Consider the definition $f = \mathit{sum} \cdot \mathit{descend}$, synthesise a recursive definition of f.
Ans:
Case 0:

$$\begin{array}{ll} & f\ 0 \\ = & \quad \{\ \text{def. of } f\ \} \\ & \mathit{sum}\ (\mathit{descend}\ 0) \\ = & \quad \{\ \text{def. of } \mathit{descend}\ \} \\ & \mathit{sum}\ [\,] \\ = & \quad \{\ \text{def. of } \mathit{sum}\ \} \\ & 0 \end{array}$$

Case $n+1$:

$$\begin{array}{ll} & f\ (n+1) \\ = & \quad \{\ \text{def. of } f\ \} \\ & \mathit{sum}\ (\mathit{descend}\ (n+1)) \\ = & \quad \{\ \text{def. of } \mathit{descend}\ \} \\ & \mathit{sum}\ ((n+1){:}\mathit{descend}\ n) \end{array}$$

$$
\begin{array}{ll}
= & \quad \{ \text{ def. of } sum \ \} \\
& (n+1) + sum\,(descend\ n) \\
= & \quad \{ \text{ def. of } f \ \} \\
& (n+1) + f\ n
\end{array}
$$

Thus we have constructed:

$$
\begin{array}{lcl}
f\,0 & = & 0 \\
f\,(n+1) & = & n + 1 + f\ n.
\end{array}
$$

2. Recall the datatype definition for internally labelled binary trees:

$$\textbf{data}\ ITree\,\alpha \quad = \quad Null \mid Node\,\alpha\,(ITree\,\alpha)\,(ITree\,\alpha).$$

(a) Consider the function *mapiTree* defined below:

$$
\begin{array}{l}
mapiTree\,f\ Null = Null, \\
mapiTree\,f\,(Node\ x\ t\ u) = \\
\qquad Node\,(f\ x)\,(mapiTree\,f\ t)\,(mapiTree\,f\ u).
\end{array}
$$

What does this function do?

Ans:

The function call $mapiTree\,f\ t$ applies f to every element in t.

(b) Define a function *sumiTree* computing the sum of all node values in an *iTree*.

Ans:

$$
\begin{array}{lcl}
sumiTree\ Null & = & 0 \\
sumiTree\,(Node\ a\ t\ u) & = & a + sumiTree\ t + sumiTree\ u
\end{array}
$$

(c) The function $one\ x = 1$ returns 1, whatever the input is. The function *sizeiTree* is specified by:

$$sizeiTree \quad = \quad sumiTree \cdot mapiTree\ one.$$

What does this function do? Derive a definition of *sizeiTree* which does not construct an intermediate tree.

Ans:

The function call $sizeiTree\ t$ computes the size of t.

Case Null:

$$
\begin{array}{ll}
& sizeiTree\ Null \\
= & \quad \{ \text{ def. of } sizeiTree \ \} \\
& sumiTree\,(mapiTree\ one\ Null) \\
= & \quad \{ \text{ def. of } mapiTree \ \} \\
& sumiTree\ Null \\
= & \quad \{ \text{ def. of } sumiTree \ \}
\end{array}
$$

Case $Node\ a\ t\ u$:

$$
\begin{array}{ll}
 & sizeiTree\ (Node\ a\ t\ u) \\
= & \quad \{\ \text{def. of } sizeiTree\ \} \\
 & sumiTree\ (mapiTree\ one\ (Node\ a\ t\ u)) \\
= & \quad \{\ \text{def. of } mapiTree\ \} \\
 & sumiTree\ (Node\ (one\ a)\ (mapiTree\ one\ u)\ (mapiTree\ one\ t)) \\
= & \quad \{\ \text{def. of } sumiTree \text{ and } one\ \} \\
 & 1 + sumiTree \cdot mapiTree\ one\ u + sumiTree \cdot mapiTree\ one\ t \\
= & \quad \{\ \text{def. of } sizeiTree\ \} \\
 & 1 + sizeiTree\ u + sizeiTree\ t
\end{array}
$$

Therefore,

$$
\begin{array}{lcl}
sizeiTree\ Null & = & 0 \\
sizeiTree\ (Node\ a\ t\ u) & = & 1 + sizeiTree\ u + sizeiTree\ t.
\end{array}
$$

3. Recall the datatype definition for externally labelled binary trees:

$$\mathbf{data}\ ETree\ \alpha \quad = \quad Tip\ \alpha \mid Bin\ (ETree\ \alpha)\ (ETree\ \alpha).$$

(a) What does this function do?

$$
\begin{array}{lcl}
mineTree\ (Tip\ x) & = & x \\
mineTree\ (Bin\ t\ u) & = & mineTree\ t \downarrow mineTree\ u
\end{array}
$$

Ans:
The function call $mineTree\ t$ returns the minimum element in t.

(b) What does this function do?

$$
\begin{array}{lcl}
repeTree\ x\ (Tip\ y) & = & Tip\ x \\
repeTree\ x\ (Bin\ t\ u) & = & Bin\ (repeTree\ x\ t)\ (repeTree\ x\ u)
\end{array}
$$

Ans:
The function call $repeTree\ x\ t$ replaces every element in t by x.

(c) What does this function do?

$$
\begin{array}{lcl}
repbymin\ t & = & \mathbf{let}\ m = mineTree\ t \\
 & & \mathbf{in}\ repeTree\ m\ t
\end{array}
$$

How many times does this program traverse the input tree?
Ans:
The function call $repbymin\ t$ replaces every element in t by its minimum element. It traverses t twice, once for computing the minimum element, once for the replacement.

(d) Consider this definition:

$$repmin\ x\ t \quad = \quad (repeTree\ x\ t, mineTree\ t)$$

Construct a recursive definition of *repmin* that traverses the tree only once.

Ans:

Case $Tip\ y$:

$$\begin{array}{ll} & repmin\ x\ (Tip\ y) \\ = & \quad \{\ \text{def. of } repmin\ \} \\ & (repeTree\ x\ (Tip\ y), mineTree\ (Tip\ y)) \\ = & \quad \{\ \text{def. of } repeTree \text{ and } mineTree\ \} \\ & (Tip\ x, y) \end{array}$$

Case $Bin\ t\ u$:

$$\begin{array}{ll} & repmin\ x\ (Bin\ t\ u) \\ = & \quad \{\ \text{def. of } repmin\ \} \\ & (repeTree\ x\ (Bin\ t\ u), mineTree\ (Bin\ t\ u)) \\ = & \quad \{\ \text{def. of } repeTree \text{ and } mineTree\ \} \\ & (Bin\ (repeTree\ x\ t)\ (repeTree\ x\ u), mineTree\ t \downarrow mineTree\ u) \\ = & \quad \{\ \text{giving names to some sub-expressions}\ \} \\ & \textbf{let}\ (t', y) = (repeTree\ x\ t, mineTree\ t) \\ & \quad\ \ (u', z) = (repeTree\ x\ u, mineTree\ u) \\ & \textbf{in}\ (Bin\ t'\ u', y \downarrow z) \\ = & \quad \{\ \text{def. of } repmin\ \} \\ & \textbf{let}\ (t', y) = repmin\ x\ t \\ & \quad\ \ (u', z) = repmin\ x\ u \\ & \textbf{in}\ (Bin\ t'\ u', y \downarrow z) \end{array}$$

Thus we have derived the definition:

$$\begin{array}{lcl} repmin\ x\ (Tip\ y) & = & (Tip\ x, y) \\ repmin\ x\ (Bin\ t\ u) & = & \textbf{let}\ (t', y) = repmin\ x\ t \\ & & \quad\ \ (u', z) = repmin\ x\ u \\ & & \textbf{in}\ (Bin\ t'\ u', y \downarrow z) \end{array}$$

(e) Redefine *repbymin* as:

$$\begin{array}{lcl} repbymin'\ t & = & \textbf{let}\ (t', m) = repmin\ m\ t \\ & & \textbf{in}\ t'. \end{array}$$

How many times does this definition of *repbymin* traverse the tree?

Ans:

The function call $repbymin\ t$ traverses t only once. The replacement is done during searching for the minimum element.

1.2 Proof by Induction

1. Prove $(xs +\!\!+ ys) +\!\!+ zs = xs +\!\!+ (ys +\!\!+ zs)$. Hint: induction on xs.

   Ans:

Case $[\,]$:

$$
\begin{array}{ll}
 & (([\,] +\!\!+\, ys) +\!\!+\, zs \\
= & \quad \{ \text{ def. of } (+\!\!+) \ \} \\
 & ys +\!\!+\, zs \\
= & \quad \{ \text{ def. of } (+\!\!+) \ \} \\
 & [\,] +\!\!+\, (ys +\!\!+\, zs)
\end{array}
$$

Case $x{:}xs$: Suppose the equality hold for xs,

$$
\begin{array}{ll}
 & ((x{:}xs) +\!\!+\, ys) +\!\!+\, zs \\
= & \quad \{ \text{ def. of } (+\!\!+) \ \} \\
 & (x{:}(xs +\!\!+\, ys)) +\!\!+\, zs \\
= & \quad \{ \text{ def. of } (+\!\!+) \ \} \\
 & x{:}((xs +\!\!+\, ys) +\!\!+\, zs) \\
= & \quad \{ \text{ induction hypothesis } \} \\
 & x{:}(xs +\!\!+\, (ys +\!\!+\, zs)) \\
= & \quad \{ \text{ definition of } (+\!\!+) \ \} \\
 & (x{:}xs) +\!\!+\, (ys +\!\!+\, zs)
\end{array}
$$

2. The function *concat* concatenates a list of lists:

$$
\begin{array}{lcl}
concat\,[\,] & = & [\,], \\
concat\,(xs : xss) & = & xs +\!\!+\, concat\,xss.
\end{array}
$$

E.g. $concat\,[[1,2],[3,4],[5]] = [1,2,3,4,5]$. Prove that:

$$sum \cdot concat = sum \cdot map\,sum.$$

Hint: you may need one of the properties proved in the lecture.
Ans:
Case $[\,]$:

$$
\begin{array}{ll}
 & (sum \cdot concat)\,[\,] \\
= & \quad \{ \text{ def. of } (\cdot) \ \} \\
 & sum(concat\,[\,]) \\
= & \quad \{ \text{ def. of } concat \ \} \\
 & sum\,[\,] \\
= & \quad \{ \text{ def. of } map \ \} \\
 & sum(map\,sum[\,]) \\
= & \quad \{ \text{ def. of } (\cdot) \ \} \\
 & (sum \cdot map\,sum)\,[\,]
\end{array}
$$

Case $xs{:}xss$: suppose the equality holds for xss,

$$
\begin{aligned}
& (sum \cdot concat)\,(xs{:}xss) \\
= \quad & \{ \text{ def. of } (\cdot) \ \} \\
& sum\,(concat\,(xs{:}xss)) \\
= \quad & \{ \text{ def. of } concat \ \} \\
& sum\,(xs +\!\!+ concat\,xss) \\
= \quad & \{ \text{ since } sum\,(xs +\!\!+ ys) = sum\,xs + sum\,ys \ \} \\
& sum\,xs + sum\,(concat\,xs) \\
= \quad & \{ \text{ induction hypothesis } \} \\
& sum\,xs + sum\,(map\,sum\,xss) \\
= \quad & \{ \text{ def. of } sum \ \} \\
& sum\,((sum\,xs){:}map\,sum\,xss) \\
= \quad & \{ \text{ def. of } map \ \} \\
& sum\,(map\,sum\,(xs{:}xss)) \\
= \quad & \{ \text{ def. of } (\cdot) \ \} \\
& (sum \cdot map\,sum)\,(xs{:}xss)
\end{aligned}
$$

3. Prove that $map\,f \cdot map\,g = map\,(f \cdot g)$.
   Ans:
   The case for [] is trivial. For the inductive case, suppose the equality holds for xs. We reason:

$$
\begin{aligned}
& (map\,f \cdot map\,g)\,(x{:}xs) \\
= \quad & \{ \text{ def. of } (\cdot) \ \} \\
& map\,f\,(map\,g\,(x{:}xs)) \\
= \quad & \{ \text{ def. of } map \ \} \\
& map\,f\,((g\,x){:}map\,g\,xs) \\
= \quad & \{ \text{ def. of } map \ \} \\
& (f\,(g\,x)){:}map\,f\,(map\,g\,xs) \\
= \quad & \{ \text{ def. of } (\cdot) \ \} \\
& ((f \cdot g)\,x){:}map\,f\,(map\,g\,xs) \\
= \quad & \{ \text{ induction hypothesis } \} \\
& ((f \cdot g)\,x){:}map\,(f \cdot g)\,xs \\
= \quad & \{ \text{ def. of } map \ \} \\
& map\,(f \cdot g)\,(x{:}xs)
\end{aligned}
$$

4. The function *swapTree* is defined by:

$$\begin{array}{lcl} \mathit{swapiTree}\ \mathit{Null} & = & \mathit{Null}, \\ \mathit{swapiTree}\ (\mathit{Node}\ a\ t\ u) & = & \mathit{Node}\ a\ (\mathit{swapiTree}\ u)\ (\mathit{swapiTree}\ t). \end{array}$$

Prove that $\mathit{swapiTree}\ (\mathit{swapiTree}\ t) = t$ for all t.
Ans:
The *Null* case is easy: $\mathit{swapTree}\ (\mathit{swapTree}\ \mathit{Null}) = \mathit{swapTree}\ \mathit{Null} = \mathit{Null}$.
To prove the case for $\mathit{Node}\ a\ t\ u$, suppose the equality holds for t and u.
For any a, by definition of *swapTree*, we have:

$$\begin{array}{ll} & \mathit{swapTree}\ (\mathit{swapTree}\ (\mathit{Node}\ a\ t\ u)) \\ = & \mathit{swapTree}\ (\mathit{Node}\ a\ (\mathit{swapTree}\ u)\ (\mathit{swapTree}\ t)) \\ = & \mathit{Node}\ a\ (\mathit{swapTree}\ (\mathit{swapTree}\ t))\ (\mathit{swapTree}\ (\mathit{swapTree}\ u)) \\ = & \quad \{\ \text{induction hypothesis}\ \} \\ & \mathit{Node}\ a\ t\ u \end{array}$$

1.3 Accumulating Parameters

1. Recall the standard definition of factorial:

$$\begin{array}{lcl} \mathit{fact}\ 0 & = & 1, \\ \mathit{fact}\ (n+1) & = & (n+1) \times \mathit{fact}\ n. \end{array}$$

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

(a) Introduce $\mathit{factit}\ n\ m = \ldots$ where m is an accumulating parameter.
Ans:
We accumulate the chain of products $(n+1) \times n \times \cdots$ in the parameter m:

$$\mathit{factit}\ n\ m = m * \mathit{fact}\ n$$

(b) Express *fact* in terms of *factit*.
Ans:
$\mathit{fact}\ n = \mathit{factit}\ n\ 1$

(c) Construct a space efficient implementation of *factit*.
Ans:
Case 0:

$$\begin{array}{ll} & \mathit{factit}\ 0\ m \\ = & \quad \{\ \text{def. of } \mathit{factit}\ \} \\ & m * \mathit{fact}\ 0 \\ = & m \end{array}$$

Case $n+1$:

$$\begin{array}{ll} & \mathit{factit}\,(n+1)\,m \\ = & \quad \{\ \text{def. of } \mathit{factit}\ \} \\ & m * \mathit{fact}\,(n+1) \\ = & \quad \{\ \text{def. of } \mathit{fact}\ \} \\ & m * (n+1) * \mathit{fact}\,n \\ = & \quad \{\ \text{def. of } \mathit{factit}\ \} \\ & \mathit{factit}\,n\,(m * (n+1)) \end{array}$$

Thus,

$$\begin{array}{lcl} \mathit{factit}\,0\,m & = & m \\ \mathit{factit}\,(n+1)\,m & = & \mathit{factit}\,n\,(m * (n+1)) \end{array}$$

2. Recall the standard definition of Fibonacci:

$$\begin{array}{l} \mathit{fib}\,0 = 0 \\ \mathit{fib}\,1 = 1 \\ \mathit{fib}\,(n+2) = \mathit{fib}\,(n+1) + \mathit{fib}\,n \end{array}$$

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

(a) Given the definition $\mathit{fibit}\,n\,x\,y = \mathit{fib}\,n \times x + \mathit{fib}\,(n+1) \times y$. Express *fib* using *fibit*.
Ans:
$\mathit{fib}\,n = \mathit{fibit}\,n\,1\,0$

(b) Derive a linear-time version of *fibit*.
Ans:
Case 0:

$$\begin{array}{ll} & \mathit{fibit}\,0\,x\,y \\ = & \quad \{\ \text{def. of } \mathit{fibit}\ \} \\ & \mathit{fib}\,0 \times x + \mathit{fib}\,1 \times y \\ = & \quad \{\ \text{def. of } \mathit{fib}\ \} \\ & 0 \times x + 1 \times y \\ = & \quad \{\ \text{arithmetics}\ \} \\ & y \end{array}$$

Case $n+1$:

$$\begin{array}{ll} & \mathit{fibit}\,(n+1)\,x\,y \\ = & \quad \{\ \text{def. of } \mathit{fibit}\ \} \\ & \mathit{fib}\,(n+1) \times x + \mathit{fib}\,(n+2) \times y \\ = & \quad \{\ \text{def. of } \mathit{fib}\ \} \\ & \mathit{fib}\,(n+1) \times x + (\mathit{fib}\,(n+1) + \mathit{fib}\,n) \times y \end{array}$$

$$
\begin{array}{ll}
= & \{ \text{ arithmetics } \} \\
& \mathit{fib}\ n \times y + \mathit{fib}\ (n+1) \times (x+y) \\
= & \{ \text{ def. of } \mathit{fibit} \ \} \\
& \mathit{fibit}\ n\ y\ (x+y)
\end{array}
$$

Therefore,

$$
\begin{array}{lcl}
\mathit{fibit}\ 0\ x\ y & = & y \\
\mathit{fibit}\ (n+1)\ x\ y & = & \mathit{fibit}\ n\ y\ (x+y)
\end{array}
$$

2 Take-Home Exercise (Due Date: July 10th)

You need to complete only one of the two exercises. Exercise 1 is worth 35 points while exercise 2 is worth 40 points.

1. Given an *iTree*, the following function *flatten* returns a list of all labels in the tree, in left-to-right order:

$$
\begin{array}{lcl}
\mathit{flatten}\ \mathit{Null} & = & [\,], \\
\mathit{flatten}\ (\mathit{Node}\ x\ t\ u) & = & \mathit{flatten}\ t +\!\!+ [x] +\!\!+ \mathit{flatten}\ u.
\end{array}
$$

   Unfortunately, *flatten* is slow. Let us try to improve it. Introduce $\mathit{flatcat}\ t\ xs = \mathit{flatten}\ t +\!\!+ xs$.

   (a) Express *flatten* in terms of *flatcat*.

   (b) Construct an efficient implementation of *flatten*. You will need some properties of $(+\!\!+)$ proved in one of the exercises.

   Hint:

   (a) To see the specification running, load `mu-code.hs` into `Hugs` or `GHCi`, and try `flatten testTree1 1`. Run your derived program to check whether it produces the same output as the specification.

   (b) The derivation works in a way similar to how *revcat* was constructed in the class. You may need to perform some steps more than once.

2. This problem considers labelling an internally-labelled binary tree:

$$
\mathbf{data}\ \mathit{iTree}\ \alpha \quad = \quad \mathit{Null} \mid \mathit{Node}\ \alpha\ (\mathit{iTree}\ \alpha)\ (\mathit{iTree}\ \alpha).
$$

   Given such a tree, for example (the labels in the tree does not matter, so let us assume they are just ()):

$$
\begin{array}{lcl}
t & = & \mathit{Node}\ ()\ (\mathit{Node}\ ()\ (\mathit{Node}\ ()\ \mathit{Null}\ \mathit{Null}) \\
& & \qquad\qquad (\mathit{Node}\ ()\ \mathit{Null}\ \mathit{Null})) \\
& & \qquad (\mathit{Node}\ ()\ \mathit{Null} \\
& & \qquad\qquad (\mathit{Node}\ ()\ (\mathit{Node}\ ()\ \mathit{Null}\ \mathit{Null}) \\
& & \qquad\qquad\qquad \mathit{Null})),
\end{array}
$$

the task is to number the nodes, in depth-first order:

$$\begin{array}{lcl} t & = & \mathit{Node}\,1\,(\mathit{Node}\,2\,(\mathit{Node}\,3\,\mathit{Null}\,\mathit{Null}) \\ & & \qquad\qquad (\mathit{Node}\,4\,\mathit{Null}\,\mathit{Null})) \\ & & \quad (\mathit{Node}\,5\,\mathit{Null} \\ & & \qquad\qquad (\mathit{Node}\,6\,(\mathit{Node}\,7\,\mathit{Null}\,\mathit{Null}) \\ & & \qquad\qquad\qquad \mathit{Null})). \end{array}$$

The following function *label* specifies how to label a tree, starting from a given initial number n:

$$\begin{array}{lcl} \mathit{label}\,\mathit{Null}\,n & = & \mathit{Null}, \\ \mathit{label}\,(\mathit{Node}\,\_\,t\,u)\,n & = & \mathit{Node}\,n\,(\mathit{label}\,t\,(1+n)) \\ & & \qquad\qquad (\mathit{label}\,u\,(1+n+\mathit{sizeiTree}\,t)), \end{array}$$

where *size* is defined by:

$$\begin{array}{lcl} \mathit{sizeiTree}\,\mathit{Null} & = & 0, \\ \mathit{sizeiTree}\,(\mathit{Node}\,x\,t\,u) & = & 1+\mathit{sizeiTree}\,t+\mathit{sizeiTree}\,u. \end{array}$$

Due to repeated call to *size*, the above definition of *label* is rather inefficient. Define:

$$\mathit{labeltl}\,t\,n \;=\; (\mathit{label}\,t\,n, n+\mathit{size}\,t),$$

derive a recursive definition for *labeltl* that runs in time linear to the size of the tree. Hint:

(a) To see the specification running, load `mu-code.hs` into `Hugs` or `GHCi`, and try `label testTree2 1`. Run your derived program to check whether it produces the same output as the specification.

(b) *labeltl* may need to call itself more than once in the recursive definition. You may need to introduce **let** in the definition, perhaps more than once.