Declarative Programming Techniques

Annalise Tarhan

April 28, 2020

1 Absolute Value of Real Numbers

```
fun {Abs X} if X<0 then ~X else X end end
```

1.1 The above function, which should calculate the absolute value of any real number, does not work. Why?

The function works correctly for integers, but not for floats, since the input value is compared to 0. To fix it, change $X < 0$ to $X < 0.0$ and require input values to be floats.

2 Cube Roots

2.1 Write a declarative program to calculate cube roots using Newton's method.

```
fun {CubeRoot X}
      fun {Improve Guess}
            (X/(Guess*Guess) + 2.0*Guess)/3.0
      end
      fun {Abs Y}
            if Y<0.0 then ~Y else Y end
      end
      fun {GoodEnough Guess}
            {Abs X-Guess*Guess*Guess}/X < 0.00001
      end
      fun {CubeRootIter Guess}
            if {GoodEnough Guess} then Guess
            else {CubeRootIter {Improve Guess}}
            end
      end
      Guess=1.0
```

```
in
      {CubeRootIter Guess}
end
```

3 The Half-Interval Method

3.1 Write a declarative program to find a root of a function f given two points a and b where $f(a) < 0 < f(b)$ using the techniques of iterative computation.

```
fun {RootFinder F A B}
      fun {RootIter Lower Upper}
            Midpoint=((Upper-Lower)/2.0 + Lower)
            fun {Abs X}
                  if X<0.0 then ~X else X end
            end
            fun {GoodEnough}
                  {Abs {F Midpoint}} < 0.00001
            end
            fun {Improve}
                  if {F Midpoint} > 0.0
                  then {RootIter Lower Midpoint}
                  else {RootIter Midpoint Upper}
                  end
            end
      in
            if {GoodEnough}
            then Midpoint
            else {Improve}
            end
      end
in
      {RootIter A B}
end
```

4 Iterative Factorial

4.1 Using the technique of state transformations from an initial state, give a definition of factorial which results in an iterative computation.

```
fun {Fact Num}
    fun {FactIter N Acc}
        if N==0 then Acc
        else {FactIter N-1 Acc*N}
        end
    end
in
    {FactIter Num 1}
end
```

5 An Iterative *SumList*

5.1 Rewrite the function *SumList* to be iterative.

```
fun {SumList Lst}
    fun {SumListIter L Acc}
        case L
        of nil then Acc
        [] Head|Tail then {SumListIter Tail Acc+Head}
        end
    end
in
    {SumListIter Lst 0}
end
```

6 State Invariants

6.1 What is a state invariant for the *IterReverse* function?

$P((Rs, Ys)) \equiv (\{Append\ \{Reverse\ Rs\}\ Ys\} = Xs)$

7 Another Append Function

```
fun {Append Ls Ms}
    case Ms
    of nil then Ls
    [] X|Mr then {Append {Append Ls [X]} Mr}
    end
end
```

7.1 Is this program correct? Does it terminate? Why or why not?

No, it is not correct, because it never reaches the base case and therefore does not terminate. {*Append Ls* [*X*]} gets stuck in a loop because unless Ms was an empty list to begin with, the case statement will always match it with $X|Mr$ where Mr is nil and add another {*Append Ls* [*X*]} to the call stack.

8 An Iterative Append

8.1 Write an iterative *append* function by defining an iterative list reversal, then an iterative function that appends the reverse of a list to another list.

```
fun {AppendIter List1 List2}
    fun {Reverse Original Reversed}
        case Original
        of nil then Reversed
        [] Head|Tail then {Reverse Tail Head|Reversed}
        end
    end
    fun {AppendReversed ReversedPart SecondPart}
        case ReversedPart
        of nil then SecondPart
        [] Head|Tail
            then {AppendReversed Tail Head|SecondPart}
        end
    end
in
    {AppendReversed {Reverse List1 nil} List2}
end
```

9 Iterative Computations and Dataflow Variables

9.1 For any iterative operation defined with dataflow variables, is it possible to give another iterative definition of the same operation that does not use dataflow variables?

Yes. Dataflow variables are used as placeholders, which allows for last call optimization, but they are not necessary. There will always be a way to rewrite iterative functions that use dataflow variables to use values only.

10 Checking If Something is a List

```
fun {Leaf X}
      case X of _|_ then false else true end
end
```

10.1 What goes wrong if the function below is used instead?

```
fun {Leaf X}
      X\=(_|_) end
end
```

Unlike case statements, which match patterns, entailment and disentailment checks match values. The problem is that disentailment checks block until it is known whether or not the two values are equal. The disentailment check performs properly when it is impossible for the input to match (_|_), but since the two _ identifiers are unbound, it will block indefinitely on anything else.

11 Limitations of Difference Lists

11.1 What goes wrong when trying to append the same difference list more than once?

What makes difference lists useful is that when the second list is an unbound variable, another list can be appended to it in constant time. However, that can only happen once. After the second list is bound, it is like any other bound variable and cannot be reassigned.

12 Complexity of List Flattening

12.1 Calculate the number of operations needed by the two versions of the *Flatten* function.

```
fun {Flatten Xs}
      case Xs
      of nil then nil
      [] X|Xr andthen {IsList X} then
           {Append {Flatten X} {Flatten Xr}}
      [] X|Xr then
           X|{Flatten Xr}
      end
end
```

The first *Flatten* function consists of a *case* statement, which means the number of operations it needs is $k_1 + max(T(s_1), T(s_2), T(s_3))$ where s_1, s_2, and s_3 are the three branches. The number of operations in the first branch, *of nil then nil* is a constant, k_2.

The number of operations in the second branch is more complicated. *IsList* is simply an if/else statement with a nested case statement. This is a constant number of operations, k_3. The number of statements in *Append* is proportional to the size of *Flatten X* plus a constant, k_4. Leave the calls to *Flatten* as $T_{Flatten}(X)$ and $T_{Flatten}(Y)$ where $X + Y = N$.

The number of operations in the third branch is a constant, k_5 plus a call to *Flatten*, $T_{Flatten}(N - 1)$. Because a call to *Append* is guaranteed to add more operations than a simple *cons* operation, we know that the second branch takes the longest.

With that information, we can refine the number of operations to $k_1 + T(s_2)$, or $k_1 + k_3 + k_4 * X + T_{Flatten}(X) + T_{Flatten}(Y)$ where $X + Y = N$. Ignoring the constants, we get a result of $X + T_{Flatten}(X) + T_{Flatten}(N - X)$ operations for a call to $Flatten(N)$.

```
fun {FlattenD Xs E}
      case Xs
      of nil then E
      [] X|Xr andthen {IsList X} then
           {FlattenD X {FlattenD Xr E}}
      [] X|Xr then
           X|{FlattenD Xr E}
      end
end
```

The structure of *FlattenD* is very similar to *Flatten*. The main difference is that instead of a call to *Append* and two calls to *Flatten*, the second branch only makes two calls to $FlattenD$. Ignoring constants, this gives a result of $T_{FlattenD}(X) + T_{FlattenD}(N - X)$ operations per call to $FlattenD(N)$.

12.2 With n elements and a maximal nesting depth k, what is the worst-case complexity of each version?

The worst-case scenario input is a list where each element is in its own single-element, deeply nested list. For example, a worst-case input list with 3 elements and maximal nesting depth of 4 would be [[[[[1]]]], [[[[2]]]], [[[[3]]]]]. The worst-case complexity of the first version would be $n^2 + n * k$. The worst-case complexity of the second version would be $n * k$. The difference is the expensive call to *Append* at each iteration.

13 Matrix Operations

13.1 Assuming that matrices are represented as lists of lists of integers, define functions to do standard matrix operations such as matrix transposition and matrix multiplication.

```
fun {MatrixAddition M1 M2}
    fun {RowAddition R1 R2}
        case R1
        of nil then nil
        [] Item|Rest then
            (R1.1+R2.1)|{RowAddition R1.2 R2.2}
        end
    end
in
    case M1
    of nil then nil
    [] Row|Rest then
        {RowAddition M1.1 M2.1}
        |{MatrixAddition M1.2 M2.2}
    end
end
```

```
fun {ScalarMultiplication Matrix N}
    fun {RowMultiplication Row N}
        case Row
        of nil then nil
        [] Item|Rest then (Item*N)
            |{RowMultiplication Rest N}
        end
```

```
    end
in
    case Matrix
    of nil then nil
    [] Row|Rest then
        {RowMultiplication Row N}
        |{ScalarMultiplication Rest N}
    end
end
```

```
fun {Transposition Matrix}
    fun {FirstColumn M}
        case M
        of nil then nil
        [] First|Rest then
            case First
            of nil then nil
            [] Head|_ then
                Head|{FirstColumn Rest}
            end
        end
    end
    fun {RestOfColumns M}
        case M
        of nil then nil
        [] First|Rest then
            case First
            of nil then nil
            [] _|Tail then
                Tail|{RestOfColumns Rest}
            end
        end
    end
in
    case Matrix
    of nil then nil
    [] _|_ then
        case {RestOfColumns Matrix}
        of nil then {FirstColumn Matrix}
        [] _ then
            {FirstColumn Matrix}
            |{Transposition {RestOfColumns Matrix}}
        end
    end
end
```

```
fun {MatrixMultiplication M1 M2}
    fun {CalculateElement Row Column}
        case Row
        of nil then 0
        [] First|Rest then
            case Column
            of nil then nil
            [] F|R then
                First*F+{CalculateElement Rest R}
            [] X then First*X
            end
        end
    end
    fun {CalculateRow Row M}
        case M
        of nil then nil
        [] _|_ then
            case {RestOfColumns M}
            of nil then
                {CalculateElement Row {FirstColumn M}}
            [] _ then
                {CalculateElement Row {FirstColumn M}}
                |{CalculateRow Row {RestOfColumns M}}
            end
        end
    end
in
    case M1
    of nil then nil
    [] First|Rest then
        {CalculateRow First M2}
        |{MatrixMultiplication Rest M2}
    end
end
```

14 FIFO Queues

14.1 What happens if you delete an element from an empty queue?

Deleting an element from an empty queue results in a failure.

14.2 What is wrong with this definition of *IsEmpty*?

```
fun {IsEmpty q(N S E)} S==E end
```

This version of *IsEmpty* only works correctly if the queue actually is empty. Otherwise, E is unbound, so the call suspends indefinitely.

15 Quicksort

15.1 Write a program to implement QuickSort using difference lists.

```
fun {Quicksort L}
    local
        fun {GetSmaller Pivot List}
            case List
            of nil then nil
            [] Head|Tail then
                if (Head<Pivot) then
                    Head|{GetSmaller Pivot Tail}
                else {GetSmaller Pivot Tail}
                end
            end
        end
        fun {GetGreater Pivot List}
            case List
            of nil then nil
            [] Head|Tail then
                if (Head>=Pivot) then
                    Head|{GetGreater Pivot Tail}
                else {GetGreater Pivot Tail}
                end
            end
        end
        fun {Append List1 List2}
            Start1#End1=List1
            Start2#End2=List2
        in
```

```
                End1=Start2
                Start1#End2
            end
            fun {QuicksortHelper List}
                case List
                of Head#Tail andthen Tail==nil then Head
                [] Head|Tail then
                    local
                        Smaller Greater
                        SortedSmaller SortedGreater in

                        Smaller={GetSmaller Head Tail}
                        Greater={GetGreater Head Tail}
                        SortedSmaller=
                            {QuicksortHelper Smaller}
                        SortedGreater=
                            {QuicksortHelper Greater}
                        {Append SortedSmaller
                            Head|SortedGreater}
                    end
                end
            end
        end
in
    {QuicksortHelper L#nil}
end
```

16 Tail-Recursive Convolution (advanced)

16.1 Write a function that takes two lists $[x_1\ x_2\ ...\ x_n]$ and $[y_1\ y_2\ ...\ y_n]$ and returns their symbolic convolution $[x_1\#y_n\ x_2\#y_{n-1}\ ...x_n\#y_1]$. The function should be tail recursive and do no more than n recursive calls.

```
fun {Convolute List1 List2}
    fun {Helper L1 L2 Markers Rev}
        case L1
        of nil then nil
        [] Head|Tail andthen
        ({Length L1}-1)>{Length Markers} then
        local X in
            Head#X
            |{Helper Tail L2.2 X|Markers L2.1|Rev}
        end
        [] Head|Tail andthen
        ({Length L1}-1)=={Length Markers} then
            Head#L2.1|{Helper Tail L2.2 Markers Rev}
        [] Head|Tail then
            Markers.1=L2.1
            Head#Rev.1
            |{Helper Tail L2.2 Markers.2 Rev.2}
        end
    end
in
    {Helper List1 List2 nil nil}
end
```