Programming Languages: Functional Programming
5. Program Calculation: Work Less by Promising More

Shin-Cheng Mu

Spring 2022

National Taiwan University and Academia Sinica

Dijkstra: "The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness. But one should not first make the program and then prove its correctness, because then the requirement of providing the proof would only increase the poor programmer's burden. On the contrary: the programmer should ..."
"...[let] correctness proof and program grow hand in hand: with the choice of the structure of the correctness proof one designs a program for which this proof is applicable."

Deriving Programs from Specifications

- In functional program derivation, the specification itself is a function, albeit probably not an efficient one.
- From the specification we construct a function that equals the specification.
- The calculation is the proof.
- In the previous class to proceed by expanding and reducing the definitions, until we obtain an inductive definition of the specification.
- But that does not work all the time.
- In this lecture we review some techniques that might work for more cases.

Tupling

Steep Lists

- A *steep list* is a list in which every element is larger than the sum of those to its right:

$$\begin{aligned} &\mathit{steep} && :: \mathit{List\ Int} \rightarrow \mathit{Bool} \\ &\mathit{steep}\ [\,] && = \mathit{True} \\ &\mathit{steep}\ (x : xs) && = \mathit{steep}\ xs\ \wedge\ x > \mathit{sum}\ xs. \end{aligned}$$

- The definition above, if executed directly, is an $O(n^2)$ program. Can we do better?
- Just now we learned to construct a generalised function which takes more input. This time, we try the dual technique: to construct a function returning more results.

Generalise by Returning More

- Recall that $fst\ (a, b) = a$ and $snd\ (a, b) = b$.
- It is hard to quickly compute $steep$ alone. But if we define

$$steepsum\ xs = (steep\ xs, sum\ xs),$$

- and manage to synthesise a quick definition of $steepsum$, we can implement $steep$ by $steep = fst \cdot steepsum$.
- We again proceed by case analysis. Trivially,

$$steepsum\ [\,] = (True, 0).$$

Deriving for the Non-Empty Case

For the case for non-empty inputs:

$$steepsum\ (x : xs)$$

Deriving for the Non-Empty Case

For the case for non-empty inputs:

$$\begin{aligned} & \mathit{steepsum}\ (x : xs) \\ = \quad & \{\ \text{definition of } \mathit{steepsum}\ \} \\ & (\mathit{steep}\ (x : xs), \mathit{sum}\ (x : xs)) \end{aligned}$$

Deriving for the Non-Empty Case

For the case for non-empty inputs:

$$
\begin{array}{ll}
& \mathit{steepsum}\ (x : xs) \\
= & \quad \{\ \text{definition of } \mathit{steepsum}\ \} \\
& (\mathit{steep}\ (x : xs), \mathit{sum}\ (x : xs)) \\
= & \quad \{\ \text{definitions of } \mathit{steep} \text{ and } \mathit{sum}\ \} \\
& (\mathit{steep}\ xs \wedge x > \mathit{sum}\ xs, x + \mathit{sum}\ xs)
\end{array}
$$

Deriving for the Non-Empty Case

For the case for non-empty inputs:

$$
\begin{aligned}
& \mathit{steepsum}\ (x : xs) \\
= & \quad \{\ \text{definition of } \mathit{steepsum}\ \} \\
& (\mathit{steep}\ (x : xs), \mathit{sum}\ (x : xs)) \\
= & \quad \{\ \text{definitions of } \mathit{steep} \text{ and } \mathit{sum}\ \} \\
& (\mathit{steep}\ xs \wedge x > \mathit{sum}\ xs, x + \mathit{sum}\ xs) \\
= & \quad \{\ \text{extracting sub-expressions involving } xs\ \} \\
& \mathbf{let}\ (b, y) = (\mathit{steep}\ xs, \mathit{sum}\ xs) \\
& \mathbf{in}\ (b \wedge x > y, x + y)
\end{aligned}
$$

Deriving for the Non-Empty Case

For the case for non-empty inputs:

$$
\begin{aligned}
& \mathit{steepsum}\ (x : xs) \\
= \quad & \{\ \text{definition of } \mathit{steepsum}\ \} \\
& (\mathit{steep}\ (x : xs), \mathit{sum}\ (x : xs)) \\
= \quad & \{\ \text{definitions of } \mathit{steep} \text{ and } \mathit{sum}\ \} \\
& (\mathit{steep}\ xs \wedge x > \mathit{sum}\ xs, x + \mathit{sum}\ xs) \\
= \quad & \{\ \text{extracting sub-expressions involving } xs\ \} \\
& \textbf{let}\ (b, y) = (\mathit{steep}\ xs, \mathit{sum}\ xs) \\
& \textbf{in}\ (b \wedge x > y, x + y) \\
= \quad & \{\ \text{definition of } \mathit{steepsum}\ \} \\
& \textbf{let}\ (b, y) = \mathit{steepsum}\ xs \\
& \textbf{in}\ (b \wedge x > y, x + y).
\end{aligned}
$$

Synthesised Program

- We have thus come up with a $O(n)$ time program:

$$
\begin{aligned}
&\mathit{steep} && = \mathit{fst} \cdot \mathit{steepsum} \\
&\mathit{steepsum}\ [\,] && = (\mathit{True}, 0) \\
&\mathit{steepsum}\ (x : xs) && = \mathbf{let}\ (b, y) = \mathit{steepsum}\ xs \\
& && \quad\ \mathbf{in}\ (b \wedge x > y, x + y),
\end{aligned}
$$

- Again we observe the phenomena that a more general function is easier to implement.

Accumulating Parameters

Reversing a List

- The function *reverse* is defined by:

$$\begin{aligned} reverse\,[\,] &= [\,], \\ reverse\,(x : xs) &= reverse\,xs \mathbin{+\!\!+} [x]. \end{aligned}$$

- E.g. $reverse\,[1,2,3,4] = ((([\,] \mathbin{+\!\!+} [4]) \mathbin{+\!\!+} [3]) \mathbin{+\!\!+} [2]) \mathbin{+\!\!+} [1] = [4,3,2,1]$.
- But how about its time complexity? Since $(\mathbin{+\!\!+})$ is $O(n)$, it takes $O(n^2)$ time to revert a list this way.
- Can we make it faster?

Introducing an Accumulating Parameter

- Let us consider a generalisation of *reverse*. Define:

$$\begin{aligned} &revcat && :: [a] \to [a] \to [a] \\ &revcat\ xs\ ys && = reverse\ xs \mathbin{+\!\!+} ys. \end{aligned}$$

- If we can construct a fast implementation of *revcat*, we can implement *reverse* by:

$$reverse\ xs = revcat\ xs\ [\,].$$

Reversing a List, Base Case

Let us use our old trick. Consider the case when xs is $[\,]$:

$$revcat\ [\,]\ ys$$

Reversing a List, Base Case

Let us use our old trick. Consider the case when xs is $[\,]$:

$$
\begin{array}{ll}
 & \mathit{revcat}\ [\,]\ \mathit{ys} \\
= & \quad \{\ \text{definition of } \mathit{revcat}\ \} \\
 & \mathit{reverse}\ [\,] +\!\!+\ \mathit{ys}
\end{array}
$$

Let us use our old trick. Consider the case when xs is $[\,]$:

$$
\begin{aligned}
& \mathit{revcat}\ [\,]\ \mathit{ys} \\
= \quad & \{\ \text{definition of } \mathit{revcat}\ \} \\
& \mathit{reverse}\ [\,] \mathbin{+\!\!+} \mathit{ys} \\
= \quad & \{\ \text{definition of } \mathit{reverse}\ \} \\
& [\,] \mathbin{+\!\!+} \mathit{ys}
\end{aligned}
$$

Reversing a List, Base Case

Let us use our old trick. Consider the case when xs is $[\,]$:

$$
\begin{aligned}
& \mathit{revcat}\ [\,]\ \mathit{ys} \\
= \quad & \{\ \text{definition of } \mathit{revcat}\ \} \\
& \mathit{reverse}\ [\,] \mathbin{+\!\!+} \mathit{ys} \\
= \quad & \{\ \text{definition of } \mathit{reverse}\ \} \\
& [\,] \mathbin{+\!\!+} \mathit{ys} \\
= \quad & \{\ \text{definition of } (\mathbin{+\!\!+})\ \} \\
& \mathit{ys}.
\end{aligned}
$$

Reversing a List, Inductive Case

Case $x : xs$:

$$revcat\ (x : xs)\ ys$$

Reversing a List, Inductive Case

Case $x : xs$:

$$
\begin{aligned}
& \mathit{revcat}\ (x : xs)\ ys \\
= \quad & \{ \text{ definition of } \mathit{revcat} \ \} \\
& \mathit{reverse}\ (x : xs) \mathbin{+\!\!+} ys
\end{aligned}
$$

Reversing a List, Inductive Case

Case $x : xs$:

$$
\begin{aligned}
& \mathit{revcat}\ (x : xs)\ ys \\
= \quad & \{\ \text{definition of } \mathit{revcat}\ \} \\
& \mathit{reverse}\ (x : xs) \mathbin{+\!\!+} ys \\
= \quad & \{\ \text{definition of } \mathit{reverse}\ \} \\
& (\mathit{reverse}\ xs \mathbin{+\!\!+} [x]) \mathbin{+\!\!+} ys
\end{aligned}
$$

Reversing a List, Inductive Case

Case $x : xs$:

$$
\begin{aligned}
& \mathit{revcat}\ (x : xs)\ ys \\
= \quad & \{\ \text{definition of } \mathit{revcat}\ \} \\
& \mathit{reverse}\ (x : xs) +\!\!+ ys \\
= \quad & \{\ \text{definition of } \mathit{reverse}\ \} \\
& (\mathit{reverse}\ xs +\!\!+ [x]) +\!\!+ ys \\
= \quad & \{\ \text{since } (xs +\!\!+ ys) +\!\!+ zs = xs +\!\!+ (ys +\!\!+ zs)\ \} \\
& \mathit{reverse}\ xs +\!\!+ ([x] +\!\!+ ys)
\end{aligned}
$$

Reversing a List, Inductive Case

Case $x : xs$:

$$
\begin{aligned}
& \mathit{revcat}\ (x : xs)\ ys \\
= \quad & \{\ \text{definition of } \mathit{revcat}\ \} \\
& \mathit{reverse}\ (x : xs) \mathbin{+\!\!+} ys \\
= \quad & \{\ \text{definition of } \mathit{reverse}\ \} \\
& (\mathit{reverse}\ xs \mathbin{+\!\!+} [x]) \mathbin{+\!\!+} ys \\
= \quad & \{\ \text{since } (xs \mathbin{+\!\!+} ys) \mathbin{+\!\!+} zs = xs \mathbin{+\!\!+} (ys \mathbin{+\!\!+} zs)\ \} \\
& \mathit{reverse}\ xs \mathbin{+\!\!+} ([x] \mathbin{+\!\!+} ys) \\
= \quad & \{\ \text{definition of } \mathit{revcat}\ \} \\
& \mathit{revcat}\ xs\ (x : ys).
\end{aligned}
$$

Linear-Time List Reversal

- We have therefore constructed an implementation of *revcat* which runs in linear time!

$$\begin{aligned} \mathit{revcat}\ [\,]\ \mathit{ys} &= \mathit{ys} \\ \mathit{revcat}\ (x : \mathit{xs})\ \mathit{ys} &= \mathit{revcat}\ \mathit{xs}\ (x : \mathit{ys}). \end{aligned}$$

- A generalisation of *reverse* is easier to implement than *reverse* itself? How come?
- If you try to understand *revcat* operationally, it is not difficult to see how it works.
    - The partially reverted list is *accumulated* in ys.
    - The initial value of ys is set by $\mathit{reverse}\ \mathit{xs} = \mathit{revcat}\ \mathit{xs}\ [\,]$.
    - Hmm... it is like a *loop*, isn't it?

Tracing Reverse

$$
\begin{aligned}
& \mathit{reverse}\ [1,2,3,4] \\
= \quad & \mathit{revcat}\ [1,2,3,4]\ [\,] \\
= \quad & \mathit{revcat}\ [2,3,4]\ [1] \\
= \quad & \mathit{revcat}\ [3,4]\ [2,1] \\
= \quad & \mathit{revcat}\ [4]\ [3,2,1] \\
= \quad & \mathit{revcat}\ [\,]\ [4,3,2,1] \\
= \quad & [4,3,2,1]
\end{aligned}
$$

$$
\begin{aligned}
\mathit{reverse}\ \mathit{xs} &= \mathit{revcat}\ \mathit{xs}\ [\,] \\
\mathit{revcat}\ [\,]\ \mathit{ys} &= \mathit{ys} \\
\mathit{revcat}\ (x : \mathit{xs})\ \mathit{ys} &= \mathit{revcat}\ \mathit{xs}\ (x : \mathit{ys})
\end{aligned}
$$

$\mathit{xs}, \mathit{ys} \leftarrow \mathit{XS}, [\,];$

while $\mathit{xs} \neq [\,]$ **do**

$\quad \mathit{xs}, \mathit{ys} \leftarrow (\mathit{tail}\ \mathit{xs}), (\mathit{head}\ \mathit{xs} : \mathit{ys});$

return ys

Tail Recursion

- Tail recursion: a special case of recursion in which the last operation is the recursive call.

$$f\,x_1\ \ldots\ x_n = \{\text{base case}\}$$
$$f\,x_1\ \ldots\ x_n = f\,x_1'\ \ldots\ x_n'$$

- To implement general recursion, we need to keep a stack of return addresses. For tail recursion, we do not need such a stack.
- Tail recursive definitions are like loops. Each x_i is updated to x_i' in the next iteration of the loop.
- The first call to f sets up the initial values of each x_i.

Accumulating Parameters

- To efficiently perform a computation (e.g. *reverse xs*), we introduce a generalisation with an extra parameter, e.g.:

$$revcat\ xs\ ys = reverse\ xs +\!\!+\ ys.$$

- Try to derive an efficient implementation of the generalised function. The extra parameter is usually used to “accumulate” some results, hence the name.
  - To make the accumulation work, we usually need some kind of associativity.
- A technique useful for, but not limited to, constructing tail-recursive definition of functions.

Accumulating Parameter: Another Example

- Recall the "sum of squares" problem:

$$\begin{aligned} sumsq\ [\,] &= 0 \\ sumsq\ (x : xs) &= square\ x + sumsq\ xs. \end{aligned}$$

- The program still takes linear space (for the stack of return addresses). Let us construct a tail recursive auxiliary function.
- Introduce $ssp\ xs\ n =$.
- Initialisation: $sumsq\ xs =$.
- Construct ssp:

Accumulating Parameter: Another Example

- Recall the "sum of squares" problem:

$$\begin{aligned} sumsq\ [\,] &= 0 \\ sumsq\ (x : xs) &= square\ x + sumsq\ xs. \end{aligned}$$

- The program still takes linear space (for the stack of return addresses). Let us construct a tail recursive auxiliary function.
- Introduce $ssp\ xs\ n = sumsq\ xs + n$.
- Initialisation: $sumsq\ xs =$.
- Construct ssp:

Accumulating Parameter: Another Example

- Recall the "sum of squares" problem:

$$\begin{aligned} &\mathit{sumsq}\ [\,] &&= 0 \\ &\mathit{sumsq}\ (x : xs) &&= \mathit{square}\ x + \mathit{sumsq}\ xs. \end{aligned}$$

- The program still takes linear space (for the stack of return addresses). Let us construct a tail recursive auxiliary function.
- Introduce $\mathit{ssp}\ xs\ n = \mathit{sumsq}\ xs + n$.
- Initialisation: $\mathit{sumsq}\ xs = \mathit{ssp}\ xs\ 0$.
- Construct ssp:

Accumulating Parameter: Another Example

- Recall the "sum of squares" problem:

$$\begin{aligned} sumsq\ [\,] &= 0 \\ sumsq\ (x : xs) &= square\ x + sumsq\ xs. \end{aligned}$$

- The program still takes linear space (for the stack of return addresses). Let us construct a tail recursive auxiliary function.
- Introduce $ssp\ xs\ n = sumsq\ xs + n$.
- Initialisation: $sumsq\ xs = ssp\ xs\ 0$.
- Construct ssp:

$$\begin{aligned} ssp\ [\,]\ n &= 0 + n \ = \ n \\ ssp\ (x : xs)\ n &= (square\ x + sumsq\ xs) + n \\ &= sumsq\ xs + (square\ x + n) \\ &= ssp\ xs\ (square\ x + n). \end{aligned}$$

Being Quicker by Doing More?

- A more generalised program can be implemented more efficiently?
  - A common phenomena! Sometimes the less general function cannot be implemented inductively at all!
  - It also often happens that a theorem needs to be generalised to be proved. We will see that later.
- An obvious question: how do we know what generalisation to pick?
- There is no easy answer — finding the right generalisation one of the most difficulty act in programming!
- For the past few examples, we choose the generalisation to exploit associativity.
- Sometimes we simply generalise by examining the form of the formula.

Labelling a List

- Consider the task of labelling elements in a list with its index.

$$index :: List\ a \rightarrow List\ (Int, a)$$
$$index = zip\ [0..]$$

- To construct an inductive definition, the case for $[\,]$ is easy. For the $x : xs$ case:

$$\begin{aligned} & index\ (x : xs) \\ = \quad & zip\ [0..]\ (x : xs) \\ = \quad & (0, x) : zip\ [1..]\ xs \end{aligned}$$

- Alas, $zip\ [1..]$ cannot be fold back to $index$!
- What if we turn the varying part into...a variable?

Labelling a List, Second Attempt

- Introduce $idxFrom :: List\ a \rightarrow Int \rightarrow List\ (Int, a)$:

$$idxFrom\ xs\ n = zip\ [n..]\ xs$$

- Initialisation: $index\ xs =$.

Labelling a List, Second Attempt

- Introduce $idxFrom :: List\ a \to Int \to List\ (Int, a)$:

$$idxFrom\ xs\ n = zip\ [n..]\ xs$$

- Initialisation: $index\ xs = idxFrom\ xs\ 0$.

- Introduce $idxFrom :: List\ a \to Int \to List\ (Int, a)$:

$$idxFrom\ xs\ n = zip\ [n..]\ xs$$

- Initialisation: $index\ xs = idxFrom\ xs\ 0$.
- We reason:

$$\begin{aligned}
& idxFrom\ (x : xs)\ n \\
= \quad & zip\ [n..]\ (x : xs) \\
= \quad & (n, x) : zip\ [n + 1..]\ xs \\
= \quad & (n, x) : idxFrom\ xs\ (n + 1)
\end{aligned}$$

Proof by Strengthening

Summing Up a List in Reverse

- Prove: $sum \cdot reverse = sum$, using the definition $reverse\ xs = revcat\ xs\ [\,]$. That is, proving $sum\ (revcat\ xs\ [\,]) = sum\ xs$.
- Base case trivial. For the case $x : xs$:

$$\begin{aligned} & sum\ (reverse\ (x : xs)) \\ = \quad & sum\ (revcat\ (x : xs)\ [\,]) \\ = \quad & sum\ (revcat\ xs\ [x]) \end{aligned}$$

- Then we are stuck, since we cannot use the induction hypothesis $sum\ (revcat\ xs\ [\,]) = sum\ xs$.
- Again, generalise $[x]$ to a variable.

Summing Up a List in Reverse, Second Attempt

- Second attempt: prove a lemma:

$$sum\ (revcat\ xs\ ys) =$$

- By letting $ys = [\,]$ we get the previous property.

Summing Up a List in Reverse, Second Attempt

- Second attempt: prove a lemma:

$$sum\ (revcat\ xs\ ys) = sum\ xs + sum\ ys$$

- By letting $ys = [\,]$ we get the previous property.

Summing Up a List in Reverse, Second Attempt

- Second attempt: prove a lemma:

$$sum\ (revcat\ xs\ ys) = sum\ xs + sum\ ys$$

- By letting $ys = [\,]$ we get the previous property.
- For the case $x : xs$ we reason:

$$sum\ (revcat\ (x : xs)\ ys)$$

Summing Up a List in Reverse, Second Attempt

- Second attempt: prove a lemma:

$$sum\ (revcat\ xs\ ys) = sum\ xs + sum\ ys$$

- By letting $ys = []$ we get the previous property.
- For the case $x : xs$ we reason:

$$\begin{aligned} & sum\ (revcat\ (x : xs)\ ys) \\ = \ & sum\ (revcat\ xs\ (x : ys)) \end{aligned}$$

Summing Up a List in Reverse, Second Attempt

- Second attempt: prove a lemma:

$$sum\ (revcat\ xs\ ys) = sum\ xs + sum\ ys$$

- By letting $ys = [\,]$ we get the previous property.
- For the case $x : xs$ we reason:

$$\begin{aligned}
& sum\ (revcat\ (x : xs)\ ys) \\
= \ & sum\ (revcat\ xs\ (x : ys)) \\
= \ & \quad \{\ \text{induction hypothesis}\ \} \\
& sum\ xs + sum\ (x : ys)
\end{aligned}$$

Summing Up a List in Reverse, Second Attempt

- Second attempt: prove a lemma:

$$sum\ (revcat\ xs\ ys) = sum\ xs + sum\ ys$$

- By letting $ys = []$ we get the previous property.
- For the case $x : xs$ we reason:

$$\begin{aligned}
& sum\ (revcat\ (x : xs)\ ys) \\
= \;& sum\ (revcat\ xs\ (x : ys)) \\
= \;& \quad \{\ \text{induction hypothesis}\ \} \\
& sum\ xs + sum\ (x : ys) \\
= \;& sum\ xs + x + sum\ ys \\
= \;& sum\ (x : xs) + sum\ ys
\end{aligned}$$

Work Less by Proving More

- A stronger theorem is easier to prove! Why is that?
- By strengthening the theorem, we also have a stronger induction hypothesis, which makes an inductive proof possible.
    - Finding the right generalisation is an art — it's got to be strong enough to help the proof, yet not too strong to be provable.
- The same with programming. By generalising a function with additional arguments, it is passed more information it may use, thus making an inductive definition possible.
    - The speeding up of *revcat*, in retrospect, is an accidental "side effect" — *revcat*, being inductive, goes through the list only once, and is therefore quicker.

A Real Case

- A property I actually had to prove for a paper:

$$\begin{aligned} & \mathit{smsp}\ (\mathit{trim}\ (x : \mathit{xs})) = \mathit{smsp}\ (\mathit{trim}\ (x : \mathit{win}\ \mathit{xs})) \\ \Leftarrow\ & \mathit{smsp}\ (\mathit{trim}\ (x : \mathit{xs})) >_d \mathit{mds}\ \mathit{xs} \end{aligned}$$

- It took me a week to construct the right generalisation:

$$\begin{aligned} & \mathit{smsp}\ (\mathit{trim}\ (\mathit{zs} \mathbin{+\!\!+} \mathit{xs})) = \mathit{smsp}\ (\mathit{trim}\ (\mathit{zs} \mathbin{+\!\!+} \mathit{win}\ \mathit{xs})) \\ \Leftarrow\ & \mathit{smsp}\ (\mathit{trim}\ (\mathit{zs} \mathbin{+\!\!+} \mathit{xs})) >_d \mathit{mds}\ \mathit{xs} \end{aligned}$$

- Once the right property is found, the actual proof was done in about 20 minutes.
- "Someone once described research as 'finding out something to find out, then finding it out'; the first part is often harder than the second."

Remark

- The $sum \cdot reverse$ example is superficial — the same property is much easier to prove using the $O(n^2)$-time definition of $reverse$.
- That's one of the reason we defer the discussion about efficiency — to prove properties of a function we sometimes prefer to roll back to a slower version.
- In our exercises there is an example where you need $revcat$ to prove a property about $reverse$.
    - Show that $reverse \cdot reverse = id$