# Programming Languages: Imperative Program Construction 12. Separation Logic II

Shin-Cheng Mu

Autumn Term, 2021

**Example: List Reversal**

- Finally we come to the canonical example: in-place list reversal.
- The aim is to come up with a program:

$$\begin{array}{l}\{\ i \text{ represents } XS\ \}\\ list\_reversal\\ \{\ j \text{ represents } reverse\ XS\ \}\end{array}$$

- But how to formally express "$i$ represents $XS$"?

## 1 Specification

**Lists**

- Well... let us quickly introduce an abstract notion of lists... and a bit of functional programming.
- $\mathbf{data}\ List = [\,] \mid Int : List$ — a list is either the empty list $[\,]$, or $x : xs$ where $x$ is $Int$ and $xs$ is a list.
- E.g $1 : 2 : 3 : [\,]$ is a list containing three items, 1, 2, and 3.
- We sometimes denote $x : [\,]$ by $[x]$, and $x : y : z : [\,]$ by $[x, y, z]$.
- Deconstructors: $head\ (1 : 2 : 3 : [\,]) = 1$, $tail\ (1 : 2 : 3 : [\,]) = 2 : 3 : [\,]$.
- For nonempty $xs$, we have $xs = head\ xs : tail\ xs$.

**Concatenation**

- Appending (concatenating) two lists:

$$\begin{array}{l}(+\!\!+) :: List \to List \to List\\ [\,] \quad +\!\!+\ ys = ys\\ (x : xs) +\!\!+\ ys = x : (xs +\!\!+\ ys)\ \ .\end{array}$$

- E.g. $[1, 2, 3] +\!\!+\ [4, 5] = [1, 2, 3, 4, 5]$.
- It can be proved that $(+\!\!+)$ is associative:

$$(xs +\!\!+\ ys) +\!\!+\ zs = xs +\!\!+\ (ys +\!\!+\ zs)\ \ .$$

**Reversal**

- List reversal:

$$\begin{array}{l}reverse :: List \to List\\ reverse\ [\,] \quad = [\,]\\ reverse\ (x : xs) = reverse\ xs +\!\!+\ [x]\ \ .\end{array}$$

- $reverse$ above is a very slow ($O(n^2)$) algorithm.

**Faster Reversal**

- One can come up with a faster algorithm using associativity.
- Let $rev\ xs\ ys = reverse\ xs +\!\!+\ ys$. The function $rev$ has a faster implemenation (which can be calculated!):

$$\begin{array}{l}rev\ [\,] \quad ys = ys\\ rev\ (x : xs)\ ys = rev\ xs\ (x : ys)\ \ .\end{array}$$

- We can then let $reverse\ xs = rev\ xs\ [\,]$.
- It actually resembles a loop...

**Representing a List**

- A list $1 : 2 : 3 : [\,]$ is an abstract entity. We have to represent it in our heap.

- By $list\ \ xs\ \ i$ we denote that "the heap represents (exactly) the list $xs$, with the first node in address $i$."

$$\begin{array}{l} list\ [\,] \qquad i \equiv \mathbf{emp} \wedge i = \mathbf{nil} \\ list\ (x : xs)\ i \equiv \langle \exists k :: (i \mapsto x, k) * list\ xs\ k \rangle \ . \end{array}$$

- Another way:

$$\begin{array}{l} list\ xs\ i \equiv (xs = [\,] \wedge \mathbf{emp} \wedge i = \mathbf{nil}) \vee \\ \quad (xs \neq [\,] \wedge \\ \quad\quad \langle \exists k :: (i \mapsto head\ xs, k) * list\ (tail\ xs)\ k \rangle) \end{array}$$

**Ghost Variables**

- Recall that in the program for fast division in Handouts 8, the variable $k$ in the following program was needed only for the proof, not for computing the result.

$$\begin{array}{l} \{A = q \times b + r \wedge 0 \leqslant r < b \wedge \\ \quad 0 \leqslant k \wedge b = 2^k \times B, bnd : b\} \\ \mathbf{do}\ b \neq B \rightarrow \\ \quad ...\ q, b, k := q \times 2, b\ /\ 2, k - 1\ ... \\ \mathbf{od} \\ \{A = q \times B + r \wedge 0 \leqslant r < B\} \end{array}$$

- $k$ was called a "ghost variable". It makes the program easier to prove (and derive). Afterwards we can remove it and use existential quantification instead:

$$\begin{array}{l} \{A = q \times b + r \wedge 0 \leqslant r < b \wedge \\ \quad \langle \exists k : 0 \leqslant k : b = 2^k \times B \rangle, bnd : b\} \\ \mathbf{do}\ b \neq B \rightarrow ...q, b := q \times 2, b\ /\ 2\ ... \\ \mathbf{od} \end{array}$$

- For this problem we will use ghost variables representing lists.

**Specification**

Problem specification:

$$\begin{array}{l} con\ XS : List \\ var\ i, j : Int \\ \{list\ XS\ i\} \\ list\_reversal \\ \{list\ (reverse\ XS)\ j\} \end{array}$$

# 2 Using Associativity

- We use our old trick — come up with a loop invariant that exploits associativity.
- Try $reverse\ XS = reverse\ xs +\!\!+ ys$.
- Initialised by $xs, ys := XS, [\,]$.
- Loop termintes when $xs = [\,]$.
- Strategy: try to shorten $xs$ in each step. Bound of loop is length of $xs$.
- Program outline:

$$\begin{array}{l} con\ XS : List \\ var\ i, j : Int, xs, ys : List \\ \{list\ XS\ i\} \\ xs, ys, j := XS, [\,], \mathbf{nil} \\ \{reverse\ XS = reverse\ xs +\!\!+ ys \wedge \\ \quad (list\ xs\ i * list\ ys\ j)\} \\ \mathbf{do}\ xs \neq [\,] \rightarrow ??? \\ \mathbf{od} \\ \{list\ (reverse\ XS)\ j\} \end{array}$$

**Loop Body**

- How do we shorten $xs$? When $xs \neq [\,]$, it can be split into head and tail.

$$\begin{array}{l} \{reverse\ XS = reverse\ xs +\!\!+ ys \wedge ..\} \\ \{reverse\ XS = reverse\ (head\ xs : tail\ xs) +\!\!+ ys \wedge ..\} \\ x, xs := head\ xs, tail\ xs \\ \{reverse\ XS = reverse\ (x : xs) +\!\!+ ys \wedge ..\} \\ \{reverse\ XS = reverse\ xs +\!\!+ (x : ys) \wedge ..\} \\ ys := x : ys \\ \{reverse\ XS = reverse\ xs +\!\!+ ys\} \end{array}$$

- Note that the last step, $ys := x : ys$, is similar to $n := 1 + n$ in other loops.
  - We try to establish the invariant for $x : ys$ (or $n + 1$), then assign $ys = x : ys$ (or $n := n + 1$) to restore the invariant.
- To justify the implication in the middle:

$$\begin{array}{ll} & reverse\ (x : xs) +\!\!+ ys \\ = & \quad \{\ \text{definition of reverse}\ \} \\ & (reverse\ xs +\!\!+ [x]) +\!\!+ ys \\ = & \quad \{\ (+\!\!+)\ \text{associative}\ \} \\ & reverse\ xs +\!\!+ ([x] +\!\!+ ys) \\ = & \quad \{\ \text{definition of}\ (+\!\!+)\ \} \\ & reverse\ xs +\!\!+ (x : ys)\ . \end{array}$$

- Similar to how we used associativity in other programs.

# 3 Pointer Manipulation

- But all these were about abstract lists. We have to update $i$ and $j$ as well.
- In the code below we omit $reverse\ XS = ..$ in the assertions and focuse on $i$ and $j$.

$$\begin{array}{l}\{list\ xs\ i * list\ ys\ j\}\\ x, xs := head\ xs, tail\ xs\\ \{list\ (x : xs)\ i * list\ ys\ j\}\\ ???\\ \{list\ xs\ i * list\ (x : ys)\ j\}\\ ys := x : ys\\ \{list\ xs\ i * list\ ys\ j\}\end{array}$$

- What to do in ????

### Shunting a Node

- Expand definitions of $list\ (x : xs)\ i * list\ ys\ j$ and $list\ xs\ i * list\ (x : ys)\ j$:

$$\begin{array}{l}\{\langle \exists k :: (i \mapsto x, k) * list\ xs\ k\rangle * list\ ys\ j\}\\ ???\\ \{list\ xs\ i * \langle \exists l :: (j \mapsto x, l) * list\ ys\ l\rangle\}\end{array}$$

- Use a lookup to remove the existential quantification:

$$\begin{array}{l}\{\langle \exists k :: (i \mapsto x, k) * list\ xs\ k\rangle * list\ ys\ j\}\\ k := {}^*(i+1)\\ \{(i \mapsto x, k) * list\ xs\ k * list\ ys\ j\}\\ ???\\ \{list\ xs\ i * \langle \exists l :: (j \mapsto x, l) * list\ ys\ l\rangle\}\end{array}$$

- Compare the pre/post-conditions, and perform some substitution:

$$\begin{array}{l}\{\langle \exists k :: (i \mapsto x, k) * list\ xs\ k\rangle * list\ ys\ j\}\\ k := {}^*(i+1)\\ \{(i \mapsto x, k) * list\ xs\ k * list\ ys\ j\}\\ ???\\ \{list\ xs\ k * \langle \exists l :: (i \mapsto x, l) * list\ ys\ l\rangle\}\\ i, j := k, i\\ \{list\ xs\ i * \langle \exists l :: (j \mapsto x, l) * list\ ys\ l\rangle\}\end{array}$$

- Guess: let $l$ be $j$:

$$\begin{array}{l}\{\langle \exists k :: (i \mapsto x, k) * list\ xs\ k\rangle * list\ ys\ j\}\\ k := {}^*(i+1)\\ \{(i \mapsto x, k) * list\ xs\ k * list\ ys\ j\}\\ ???\\ \{list\ xs\ k * (i \mapsto x, j) * list\ ys\ j\}\\ i, j := k, i\\ \{list\ xs\ i * \langle \exists l :: (j \mapsto x, l) * list\ ys\ l\rangle\}\end{array}$$

- Apparently all that's left to do is —

$$\begin{array}{l}\{\langle \exists k :: (i \mapsto x, k) * list\ xs\ k\rangle * list\ ys\ j\}\\ k := {}^*(i+1)\\ \{(i \mapsto x, k) * list\ xs\ k * list\ ys\ j\}\\ {}^*(i+1) := j\\ \{list\ xs\ k * (i \mapsto x, j) * list\ ys\ j\}\\ i, j := k, i\\ \{list\ xs\ i * \langle \exists l :: (j \mapsto x, l) * list\ ys\ l\rangle\}\end{array}$$

### Program So Far

$$\begin{array}{l}\{list\ XS\ i\}\\ xs, ys, j := XS, [\,], \mathbf{nil}\\ \{reverse\ XS = reverse\ xs +\!\!+ ys \wedge\\ \quad (list\ xs\ i * list\ ys\ j)\}\\ \mathbf{do}\ xs \neq [\,] \rightarrow\\ \quad x, xs := head\ xs, tail\ xs\\ \quad \{(list\ (x : xs)\ i * list\ ys\ j) \wedge\\ \quad\quad reverse\ XS = reverse\ (x : xs) +\!\!+ ys\}\\ \quad k := {}^*(i+1)\\ \quad {}^*(i+1) := j\\ \quad i, j := k, i\\ \quad \{(list\ xs\ i * list\ (x : ys)\ j) \wedge\\ \quad\quad reverse\ XS = reverse\ xs +\!\!+ (x : ys)\}\\ \quad ys := x : ys\\ \mathbf{od}\\ \{list\ (reverse\ XS)\ j\}\end{array}$$

### Remove Ghost Variables

- Finally, recall that we do not actually have $List$ in the executable code.
- Remove all the ghost variables.
- $xs \neq [\,]$ can be replaced by $i \neq \mathbf{nil}$.
- Final program:

$$
\begin{array}{l}
\textbf{var } i, j, k : Int \\
\{list\ XS\ i\} \\
j := \textbf{nil} \\
\{\langle \exists xs, ys :: (list\ xs\ i * list\ ys\ j) \wedge \\
\quad reverse\ XS = reverse\ xs +\!\!+ ys \rangle\} \\
\textbf{do } i \neq \textbf{nil} \rightarrow k := {}^*(i+1) \\
\qquad\qquad\quad {}^*(i+1) := j \\
\qquad\qquad\quad i, j := k, i \\
\textbf{od} \\
\{list\ (reverse\ XS)\ j\}
\end{array}
$$

# 4 Discussions

- With the ghost variables presented, it is clear that the derivation of this program follows the pattern we have been practicing:
  - construct an invariant that exploits associativity;
  - make progress by shifting some elements to the "accumulating" part;
  - the last assignment drives the loop.
- Without the ghost variable, leaving us with a less comprehensible program.
- The invariant, the bound, the hidden variables... these are what drives the development of the program. They are the foundation of the program.
- The executable code is merely derived.
- However, these foundations are often hidden in comments, removed, or forgotten. Only the executable code remains. Like flooded landscape where you see only the tips of hills.
- Programs are not supposed to be understood by reading the executable code.

**More on Separation Logic**

- We could merely touch a little bit of separation logic.
- I highly recommend Reynold's paper [Rey02] or lecture notes [Rey11] for more information.

# References


[Rey02] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In G. D. Plotkin, editor, *Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE Computer Society Press, 2002.

[Rey11] J. C. Reynolds. 15-818A3 Introduction to Separation Logic. Carnegie Mellon University. `https://www.cs.cmu.edu/afs/cs.cmu.edu/project/fox-19/member/jcr/www15818As2011/cs818A3-11.html`, 2011.