Programming Languages: Imperative Program Construction 12. Separation Logic II

Shin-Cheng Mu

Autumn Term, 2021

National Taiwan University and Academia Sinica

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

  { *i* represents *XS* }
  list_reversal
  { *j* represents *reverse XS* }

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

Specification

- Well... let us quickly introduce an abstract notion of lists... and a bit of functional programming.
- $\textbf{data}\ \mathit{List} = [\,] \mid \mathit{Int} : \mathit{List}$ — 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: $\mathit{head}\ (1 : 2 : 3 : [\,]) = 1$, $\mathit{tail}\ (1 : 2 : 3 : [\,]) = 2 : 3 : [\,]$.
- For nonempty xs, we have $xs = \mathit{head}\ xs : \mathit{tail}\ xs$.

- Appending (concatenating) two lists:

$$
\begin{aligned}
&(\mathbin{+\!\!+}) :: \mathit{List} \to \mathit{List} \to \mathit{List} \\
&[\,] \qquad\quad \mathbin{+\!\!+}\ \mathit{ys} = \mathit{ys} \\
&(x : \mathit{xs}) \mathbin{+\!\!+} \mathit{ys} = x : (\mathit{xs} \mathbin{+\!\!+} \mathit{ys}) \quad .
\end{aligned}
$$

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

$$(\mathit{xs} \mathbin{+\!\!+} \mathit{ys}) \mathbin{+\!\!+} \mathit{zs} = \mathit{xs} \mathbin{+\!\!+} (\mathit{ys} \mathbin{+\!\!+} \mathit{zs}) \quad .$$

Reversal

- List reversal:

$$\begin{aligned} &\mathit{reverse} :: \mathit{List} \to \mathit{List} \\ &\mathit{reverse}\ [\,] \qquad\quad = [\,] \\ &\mathit{reverse}\ (x : xs) = \mathit{reverse}\ xs +\!\!+ [x] \quad . \end{aligned}$$

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

- 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!):

$$
\begin{aligned}
&rev\ []\ &&ys = ys\\
&rev\ (x:xs)\ &&ys = rev\ xs\ (x:ys)\quad .
\end{aligned}
$$

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

- 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*."

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

- Another way:

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

- 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.

$$\{A = q \times b + r \wedge 0 \leqslant r < b \wedge$$
$$0 \leqslant k \wedge b = 2^k \times B, bnd : b\}$$
$$\textbf{do } b \neq B \rightarrow$$
$$\dots q, b, k := q \times 2, b \,/\, 2, k - 1 \dots$$
$$\textbf{od}$$
$$\{A = q \times B + r \wedge 0 \leqslant r < B\}$$

- 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:

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

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

Problem specification:

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

Using Associativity

- We use our old trick — come up with a loop invariant that exploits associativity.
- Try $reverse\ XS = reverse\ xs \mathbin{+\!\!+} ys$.
- 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:

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

Loop Body

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

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

- 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:

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

- Similar to how we used associativity in other programs.

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.

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

- What to do in ????

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

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

- Use a lookup to remove the existential quantification:

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

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

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

- Guess: let l be j:

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

- Apparently all that's left to do is —

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

$\{list\ XS\ i\}$

$xs, ys, j := XS, [\,], \mathbf{nil}$

$\{reverse\ XS = reverse\ xs \mathbin{+\!\!+} ys \wedge$

$\quad (list\ xs\ i * list\ ys\ j)\}$

$\mathbf{do}\ xs \neq [\,] \rightarrow$

$\quad x, xs := head\ xs, tail\ xs$

$\quad \{(list\ (x : xs)\ i * list\ ys\ j) \wedge$

$\qquad reverse\ XS = reverse\ (x : xs) \mathbin{+\!\!+} ys\}$

$\quad k := {}^{*}(i + 1)$

$\quad {}^{*}(i + 1) := j$

$\quad i, j := k, i$

$\quad \{(list\ xs\ i * list\ (x : ys)\ j) \wedge$

$\qquad reverse\ XS = reverse\ xs \mathbin{+\!\!+} (x : ys)\}$

$\quad ys := x : ys$

$\mathbf{od}$

$\{list\ (reverse\ XS)\ i\}$

- 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:

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

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.

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