Functional Correctness via Refinement

Deepak D'Souza

Department of Computer Science and Automation
Indian Institute of Science, Bangalore.

09 February 2022

Outline

Motivation for Functional Correctness

- ER models and model-checking stop short of addressing full functional correctness
- Refinement is a standard way of reasoning about functional correctness.
- Technique used is "deductive" in nature, rather than exploring reachable states.

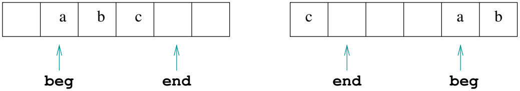
Motivating Example: C implementation of a queue

```
int A[MAXLEN];
unsigned beg,
  end, len;

void init() {
  beg = 0;
  end = 0;
  len = 0;
}

int deq() {...}

void enq(int t) {
  if (len == MAXLEN)
    assert(0);
    // exception
  A[end] = t;
  if (end < MAXLEN-1)
    end++;
  else
    end = 0;
  len++;
}
```



Motivating example: FreeRTOS

FreeRTOS Real-Time Operating System.

Extracts from code: TaskDelay()

```
void TaskDelay(portTickType xTicksToDelay){
    portTickType xTimeToWake;
    signed portBASE_TYPE xAlreadyYielded = pdFALSE;

    if( xTicksToDelay > (portTickType) 0){
        vTaskSuspendAll();
    /* Calculate the time to wake - this may overflow but this
       is not a problem. */
    xTimeToWake = xTickCount + xTicksToDelay;
    /* We must remove ourselves from the ready list before adding
       ourselves to the blocked list as the same list item is used
       for both lists. */
    vListRemove((xListItem *) &(pxCurrentTCB->xGenericListItem));
    /* The list item will be inserted in wake time order. */
    listSET_LIST_ITEM_VALUE(&(pxCurrentTCB->xGenericListItem),
                            xTimeToWake);
    ....
    portYIELD_WITHIN_API();
}
```

Abstract model of the scheduler in Z

Scheduler

$maxPrio, maxNumVal, tickCount, topReadyPriority : \mathbb{N}$
$tasks : \mathbb{P}\ TASK$
$priority : TASK \rightarrow\!\!\!\!\!\!+ \mathbb{N}$
$running\_task, idle : TASK$
$ready : \mathrm{seq}\ (\mathrm{iseq}\ TASK)$
$delayed : \mathrm{seq}\ TASK \times \mathbb{N}$
$blocked : \mathrm{seq}\ TASK$
$\ldots$

$idle \in tasks \wedge idle \in \mathrm{ran} \frown/(\mathrm{ran}\ ready)$
$running\_task \in tasks \wedge topReadyPriority \in \mathrm{dom}\ ready$
$\forall i, j : \mathrm{dom}\ delayed \mid (i < j) \bullet delayed(i).2 \leq delayed(j).2$
$\forall tcn : \mathrm{ran}\ delayed \mid tcn.2 > tickCount$
$running\_task = head\ ready(topReadyPriority)$
$\mathrm{dom}\ priority = tasks \wedge tickCount \leq maxNumVal$
$\forall i, j : \mathrm{dom}\ blocked \mid (i < j) \implies priority(blocked(i)) \geq priority(blocked(j))$
$\ldots$

Z model of `TaskDelay` operation

$$
\begin{array}{l}
\textit{TaskDelay} \\
\hline
\Delta \textit{Scheduler} \\
\textit{delay}? : \mathbb{N} \\
\textit{delayedPrefix}, \textit{delayedSuffix} : \mathrm{seq}\ \textit{TASK} \times \mathbb{N} \\
\textit{running}! : \textit{TASK} \\
\hline
\textit{delay} > 0 \land \textit{delay} \leq \textit{maxNumVal} \land \textit{running\_task} \neq \textit{idle} \\
\textit{tail ready}(\textit{topReadyPriority}) \neq \langle\rangle \land \textit{delayed} = \textit{delayedPrefix} \frown \textit{delayedSuffix} \\
\forall\, \textit{tcn} : \mathrm{ran}\ \textit{delayedPrefix} \mid \textit{tcn}.2 \leq (\textit{tickCount} + \textit{delay}?) \\
\textit{delayedSuffix} \neq \langle\rangle \implies (\textit{head delayedSuffix}).2 > (\textit{tickCount} + \textit{delay}?) \\
\textit{running\_task}' = \textit{head tail ready}(\textit{topReadyPriority}) \\
\textit{ready}' = \textit{ready} \oplus \{\ (\ \textit{topReadyPriority} \mapsto \textit{tail ready}(\textit{topReadyPriority})\ )\ \} \\
\textit{delayed}' = \textit{delayedPrefix} \frown \langle(\textit{running\_task}, (\textit{tickCount} + \textit{delay}?))\rangle \frown \textit{delayedSuffix} \\
\ldots
\end{array}
$$

Overview of plan for functional correctness

Theory

- ADTs
- Z-style refinement
  - Equivalent Refinement Condition
- Transition system based ADTs
  - ADT transition system

Tools

- Rodin
  - Models
  - Assertions
  - Proof
- VCC
  - Floyd-Hoare style annotations and proofs
  - Ghost language constructs
  - Encoding Refinement Conditions in VCC

ADT Type

An *ADT type* is a finite set N of *operation names*.

- Each operation name n in N has an associated *input type* I_n and an *output type* O_n, each of which is simply a set of values.
- We require that the set of operations N includes a designated *initialization operation* called *init*.

ADT definition

An *ADT* of type N is a structure of the form

$$\mathcal{A} = (Q, U, \{op_n\}_{n \in N})$$

where

- Q is the set of states of the ADT,
- $U \in Q$ is an arbitrary state in Q used as an *uninitialized* state,
- Each op_n is a (possibly non-deterministic) *realisation* of the operation n given by $op_n \subseteq (Q \times I_n) \times (Q \times O_n)$
- Further, we require that the *init* operation depends only on its argument and not on the originating state: thus $init(p, a) = init(q, a)$ for each $p, q \in Q$ and $a \in I_{init}$.

ADT type example: Queue

QType

ADT type $QType = \{init, enq, deq\}$ with

$$
\begin{aligned}
I_{init} &= \{nil\}, \\
O_{init} &= \{ok\}, \\
I_{enq} &= \mathbb{B}, \\
O_{enq} &= \{ok, fail\}, \\
I_{deq} &= \{nil\}, \\
O_{deq} &= \mathbb{B} \cup \{fail\}.
\end{aligned}
$$

Here $\mathbb{B}$ is the set of bit values $\{0, 1\}$, and *nil* is a "dummy" argument for the operations *init* and *deq*.

ADT example: Queue of length k of type $QType$

$QADT_k$

$QADT_k = (Q, U, \{op_n\}_{n \in QType})$ where

$$
\begin{array}{lll}
Q & = & \{\epsilon\} \cup \bigcup_{i=1}^{k} \mathbb{B}^i \\
op_{init} & \text{is given by} & op_{init}(q, nil, \epsilon, ok),\ \forall\, q \in Q \\
op_{enq} & \text{is given by} & op_{enq}(q, a, q \cdot a, ok),\ \forall\, q \in Q, a \in \mathbb{B}, |q| < k \\
op_{deq} & \text{is given by} & op_{deq}(b \cdot q, nil, q, b),\ \forall\, b \in \mathbb{B},\ b \cdot q \in Q
\end{array}
$$

Language of sequences of operation calls of an ADT

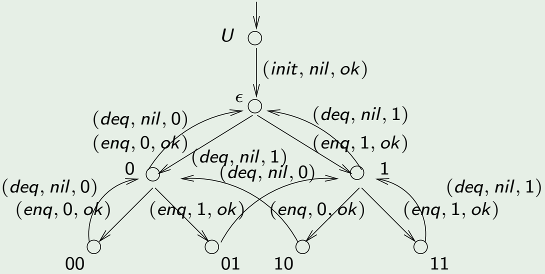
- An ADT $\mathcal{A} = (Q, U, \{op_n\}_{n \in N})$ of type N induces a (deterministic) transition system $\mathcal{S}_{\mathcal{A}} = (Q, \Sigma_N, U, \Delta)$ where
  - $\Sigma_N = \{(n, a, b) \mid n \in N, a \in I_n, b \in O_n\}$ is the set of *operation call* labels corresponding to the ADT type N. The action label (n, a, b) represents a call to operation n with input a that returns the value b.
  - Δ is given by

$$(p, (n, a, b), q) \in \Delta \text{ iff } op_n(p, a, q, b).$$

- We define the language of *initialised sequences of operation calls* of $\mathcal{A}$, denoted $L_{init}(\mathcal{A})$, to be $L(\mathcal{S}_{\mathcal{A}}) \cap ((init, -, -) \cdot \Sigma_N^*)$.

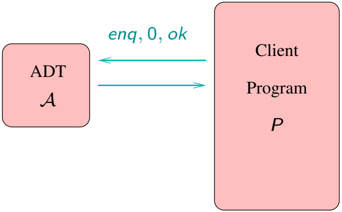
Example: Transition system induced by $QADT_2$

TS induced by $QADT_2$



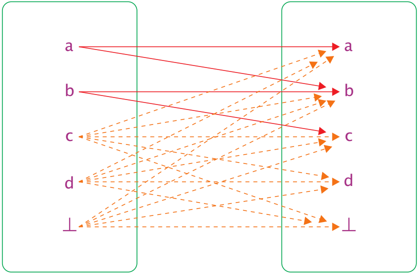
Idea behind refinement definition we will use

- A client program interacts with an ADT via a sequence of calls. If the ADT is called with an operation that is undefined in its current state, then it is assumed to "break" and return any possible value (including $\perp$); thereafter any sequence of calls/ret vals is possible.
- $L_{init}(\mathcal{A}^+)$ is the possible sequences a client of $\mathcal{A}$ can see.
- $\mathcal{B} \preceq \mathcal{A}$ iff whatever the client can see with $\mathcal{B}$, it could also have seen with $\mathcal{A}$.



This notion of refinement is from Hoare, He, Sanders et al, *Data Refinement Refined*, Oxford Univ Report, 1985.

Totalized version of a relation



$$
\begin{aligned}
R &= \{(a,a),(a,b),(b,b),(b,c)\}.\\
R^{+} &= \{(a,a),(a,b),(b,b),(b,c)\} \cup \{(c,a),(c,b),(c,c),(c,d),(c,\perp),\\
&\quad (d,a),(d,b),(d,c),(d,d),(d,\perp),(\perp,a),(\perp,b),(\perp,c),(\perp,d),(\perp,\perp)\}
\end{aligned}
$$

R^{+} adds a new element $\perp$ to domain and co-domain, and makes R total on all elements outside the domain of R.

Relation S refines relation R iff $S^{+} \subseteq R^{+}$. Thus S is "more defined" than R, and may resolve some non-determinism in R.

Totalized version of an ADT $\mathcal{A}$

Given an ADT $\mathcal{A} = (Q, U, \{op_n\}_{n \in N})$ over a data type N, define the totalized version of $\mathcal{A}$, to be an ADT $\mathcal{A}^+$ of type N^+:

$$\mathcal{A}^+ = (Q \cup \{E\}, U, \{op_n^+\}_{n \in N}), \text{ where}$$

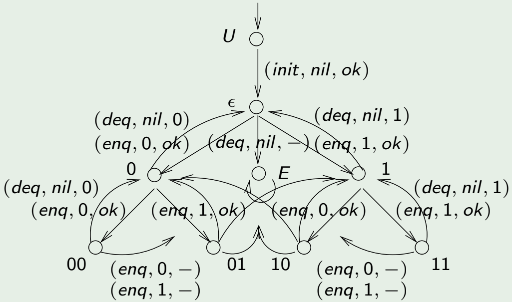
- N^+ has input type I_n and output type $O_n^+ = O_n \cup \{\perp\}$, where $\perp$ is a new output value.
- E is a new "error" state
- op_n^+ is the completed version of operation op_n, obtained as follows:
  - If $(q, a) \notin \mathrm{pre}\,(op_n)$, then add (q, a, E, b') to op_n^+ for each $b' \in O_n^+$.
  - Add $(E, a, E, b') \in op_n^+$ for each $a \in I_n$ and $b' \in O_n^+$.

Here $\mathrm{pre}\,(op_n)$ is the set of state-input pairs on which op_n is defined. Thus $(p, a) \in \mathrm{pre}\,(op_n)$ iff $\exists\, q, b$ such that $op_n(p, a, q, b)$.
If op_n is invoked outside this precondition, the data-structure is assumed to "break" and allow any possible interaction sequence after that.

$\mathcal{A}^+$ represents the interaction sequences that a client of $\mathcal{A}$ may encounter while using $\mathcal{A}$ as a data-structure.

Example: Transition system induced by $QADT_2^+$

TS induced by $QADT_2^+$



Refinement between ADTs

Let $\mathcal{A}$ and $\mathcal{B}$ be ADTs of type N. We say $\mathcal{B}$ refines $\mathcal{A}$, written

$$\mathcal{B} \preceq \mathcal{A},$$

iff

$$L_{init}(\mathcal{B}^+) \subseteq L_{init}(\mathcal{A}^+).$$

Examples of refinement:

- $QADT_3$ refines $QADT_2$.
- Let $QADT_2'$ be the version of $QADT_2$ where we check for emptiness/fullness of queue and return *fail* instead of being undefined. Then $QADT_2'$ refines $QADT_2$.

Exercise

Exercise

Is it true that

- $QADT_2$ refines $QADT_3$?
- $QADT_2$ refines $QADT_2'$?

Transitivity of refinement

It follows immediately from its definition that refinement is transitive:

Proposition

Let $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ be ADT's of type N, such that $\mathcal{C} \preceq \mathcal{B}$, and $\mathcal{B} \preceq \mathcal{A}$. Then $\mathcal{C} \preceq \mathcal{A}$.

Refinement Condition (RC)

Let $\mathcal{A} = (Q, U, \{op_n\}_{n \in N})$ and $\mathcal{A}' = (Q', U', \{op_n\}_{n \in N})$ be ADTs of type N. We give a *sufficient* condition for $\mathcal{A}'$ to refine $\mathcal{A}$, based on an "abstraction relation" that relates states of $\mathcal{A}'$ and $\mathcal{A}$.

We say $\mathcal{A}$ and $\mathcal{A}'$ satisfy condition (RC) if there exists a relation $\rho \subseteq Q' \times Q$ such that:

(init) Let $a \in I_{init}$ and let (q'_a, b) be a resultant state and output after an $init(a)$ operation in $\mathcal{A}'$. Then either $a \notin \mathrm{pre}\,(init_{\mathcal{A}})$, or there exists q_a such that $(q_a, b) \in init_{\mathcal{A}'}(a)$, with $\rho(q'_a, q_a)$.

(g-weak) For each $n \in N$, $a \in I_n$, $b \in O_n$, $p \in Q$ and $p' \in Q'$, with $(p', p) \in \rho$, if $(p, a) \in \mathrm{pre}\,(op_n)$ in $\mathcal{A}$, then $(p', a) \in \mathrm{pre}\,(op_n)$ in $\mathcal{A}'$. (guard weakening).

(sim) For each $n \in N$, $a \in I_n$, $b \in O_n$, $p \in Q$ and $p' \in Q'$, with $(p', p) \in \rho$; whenever $p' \xrightarrow{(n,a,b)} q'$ and $(p, a) \in \mathrm{pre}\,(op_n)$ in $\mathcal{A}$, then there exists $q \in Q$ such that $p \xrightarrow{(n,a,b)} q$ with $(q', q) \in \rho$.

Illustrating condition (RC)

(RC-init):

$$
\begin{tikzcd} \circ \arrow[r, "{(init, a, b)}"] & q'_a \end{tikzcd}
\Longrightarrow
\begin{tikzcd} \circ \arrow[r, "{(init, a, b)}"] & q_a \\ \circ \arrow[r, "{(init, a, b)}"] & q'_a \arrow[u, dotted, "\rho"'] \end{tikzcd}
$$

(RC-g-weak):

$$
\begin{tikzcd} p \arrow[r, "{(n, a, b)}"] & {} \\ p' \arrow[u, dashed, bend left, "\rho"] & \end{tikzcd}
\Longrightarrow
\begin{tikzcd} p \arrow[r, "{(n, a, b)}"] & {} \\ p' \arrow[u, dashed, bend left, "\rho"] \arrow[r, "{(n, a, b')}"] & {} \end{tikzcd}
$$

(RC-sim-2):

$$
\begin{tikzcd} & {} \\ p \arrow[ur, "{(n, a, b')}"] & \\ p' \arrow[u, dashed, bend left, "\rho"] \arrow[r, "{(n, a, b)}"] & q' \end{tikzcd}
\Longrightarrow
\begin{tikzcd} p \arrow[r, "{(n, a, b)}"] & q \\ p' \arrow[u, dashed, bend left, "\rho"] \arrow[r, "{(n, a, b)}"] & q' \arrow[u, dashed, bend left, "\rho"] \end{tikzcd}
$$

Exercise

Exercise

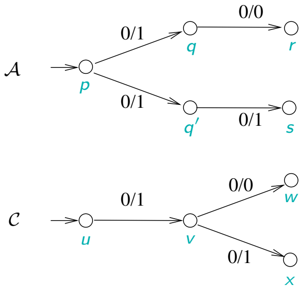
Find an abstraction relation ρ for which $QADT_2$ and $QADT_3$ satisfy condition (RC).

Condition (RC) is sufficient for refinement

If $\mathcal{A}$ and $\mathcal{C}$ are ADTs of the same type, and ρ is an abstraction relation from $\mathcal{C}$ to $\mathcal{A}$ satisfying condition (RC), then $\mathcal{C}$ refines $\mathcal{A}$.

$$
\begin{tikzcd}
\text{Abstract} & {} \arrow[r, "init(a)"] & a_0 \arrow[r, "{n, a, b}"] & a_1 \arrow[r, "{n', a', b'}"] & a_2 \arrow[r, "{n'', a'', b''}"] & a_3 \\
\text{Concrete} & {} \arrow[r, "init(a)"] & c_0 \arrow[u, dashed, bend left, "\rho"] \arrow[r, "{n, a, b}"] & c_1 \arrow[u, dashed, bend left] \arrow[r, "{n', a', b'}"] & c_2 \arrow[u, dashed, bend left] \arrow[r, "{n'', a'', b''}"] & c_3 \arrow[u, dashed, bend left]
\end{tikzcd}
$$

Example showing that RC conditions are not necessary for refinement



- $\mathcal{C}$ refines $\mathcal{A}$. In fact both $\mathcal{A}$ and $\mathcal{C}$ refine eachother, since $L_{init}(\mathcal{A}^{+}) = L_{init}(\mathcal{C}^{+})$.
- However, there is no abstraction relation ρ from $\mathcal{C}$ to $\mathcal{A}$ that satisfies the conditions (RC).

ADT Transition System

An *ADT transition system* of type N is of the form

$$\mathcal{S} = (Q_c, Q_l, \Sigma_l, U, \{\delta_n\}_{n \in N})$$

where

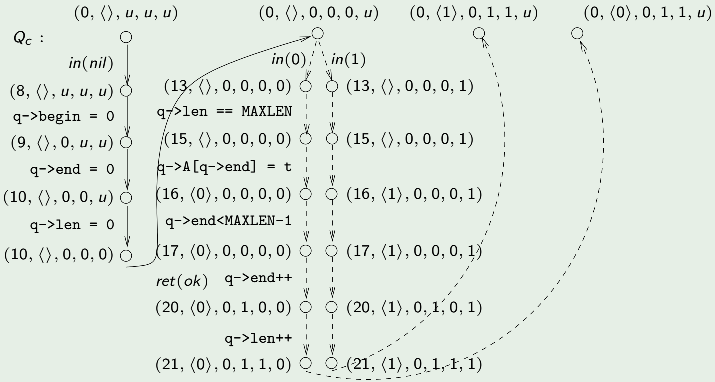
- Q_c is the set of "complete" states of the ADT (where an ADT operation is complete) and Q_l is the set of "incomplete" or "local" states of the ADT. The set of states Q of the ADT TS is the disjoint union of Q_c and Q_l.
- Σ_l is a finite set of *internal* or *local* action labels.
  - Let $\Gamma^i_N = \{in(a) \mid \ n \in N \text{ and } \ a \in I_n\}$ be the set of *input* labels corresponding to the ADT of type N. The action $in(a)$ represents reading an argument with value a.
  - Let $\Gamma^o_N = \{ret(b) \mid \ n \in N \text{ and } \ b \in O_n\}$ be the set of *return* labels corresponding to the ADT of type N. The action $ret(b)$ represents a return of the value b.
  - Let Σ be the disjoint union of Σ_l, Γ^i_N and Γ^o_N.

ADT Transition System, contd.

- For each $n \in N$, δ_n is a transition relation of the form: $\delta_n \subseteq Q \times \Sigma \times Q$, that implements the operation n. It must satisfy the following constraints:
  - it is complete for the input actions in Γ^i_N.
  - Each transition labelled by an input action in Γ^i_N begins from a Q_c state and each transition labelled by a return action in Γ^o_N ends in a Q_c state. All other transitions begin and end in a Q_l state.

Example: ADT Transition System induced by `queue.c`

Part of the ADT TS induced by `queue.c`, showing `init` and `enq` opns



ADT induced by an ADT TS

An ADT transition system like $\mathcal{S}$ above induces an ADT $\mathcal{A}_{\mathcal{S}}$ of type N given by $\mathcal{A}_{\mathcal{S}} = (Q_c, U, \{op_n\}_{n \in N})$ where for each $n \in N$, $p \in Q_c$, and $a \in I_n$, we have $op_n(p, a, q, b)$ iff there exists a path of the form $p \xrightarrow{in(a)} r_1 \xrightarrow{l_1} \cdots \xrightarrow{l_{k-1}} r_k \xrightarrow{ret(b)} q$ in $\mathcal{S}$.

We say that an ADT TS $\mathcal{S}'$ refines another ADT TS $\mathcal{S}$ iff $\mathcal{A}_{\mathcal{S}'}$ refines $\mathcal{A}_{\mathcal{S}}$.

Phrasing refinement conditions in VCC

```
typedef struct AC {
  abstract state
  invariants on abs state
  concrete state
  invariants on conc state
  gluing invariant on joint abs-conc state
} AC;

operation n(AC *p, arg a)
_(requires \wrapped(p)) // glued joint state
_(requires G)  // precondition G of abs op
_(ensures \wrapped(p)) // restores glued state
_(decreases 0) // conc op terminates whenever G is true
{
  _(unwrap p)
  // abs op body
  // conc op body
  _(wrap p)
}

init(*p)
_(ensures \wrapped(p)) {...}
```