A. Generating the Verification Condition

Assume there are \( m \) procedure calls in \( P \), then \( P \) can be divided into \( m + 1 \) basic segments (subprograms without procedure calls):

\[
S_0 \cup g_1 \cup \ldots \cup g_m
\]

where \( S_d \) is the \( d \)-th basic segment and \( g_d \) is the \( d \)-th procedure call.

For each \( d \in [m] \), let the \( d \)-th procedure call in \( P \) be the \( T_d \)-th statement (we also extend the index \( d \) to \( -1, 0 \) and \( m + 1 \) such that \( l_0 = 0 \) and \( l_{m+1} = n + 1 \)). Note that \( E \) requires that a portion of the state \( C_{T_{d-1}} \) satisfies the precondition of the call, and a portion of the state \( C_{T_d} \) satisfies the postcondition of the call. We denote the two required portions \( C_{T_{d-1}} \cup \text{Call}_d \) and \( C_{T_d} \cup \text{Return}_d \), respectively, where \( \text{Call}_d \subseteq R_{T_d} \) and \( \text{Return}_d \subseteq R_{T_d} \) are two sets of records.

Let all the location variables appearing in \( P \) be \( \text{LVars} \). We call a location variable \( v \) dereferenced if \( v \) appears on the left-hand side of a dereferencing operator "\( \& \)" in \( P \). We call a location variable \( v \) modified if \( v \) appears in a statement of the form \( v := u \) or \( v.d = j \) in \( P \). Then we can extract the set of dereferenced variables \( \text{Deref}_d \) and the set of modified variables \( \text{Mod}_d \). Note that a modified variable is always dereferenced, i.e., \( \text{Mod}_d \subseteq \text{Deref}_d \). For each basic segment \( S_d \), let the dereferenced and modified variables within the segment be \( \text{Deref}_d \) and \( \text{Mod}_d \), respectively.

For the \( d \)-th procedure call, let the pre- and post-condition associated with the procedure be \( \phi_{d}^{pre}(\vec{v}, \vec{z}, \vec{c}) \) and \( \phi_{d}^{post}(\text{ret}, \vec{v}, \vec{z}, \vec{c}) \), respectively. Since \( E \) is a normal execution, we have \( C_{T_{d-1}} \models T(\phi_{T_{d-1}}^{\text{pre}}(\vec{v}, \vec{z}, \vec{c}), \text{Call}_d) \) and \( C_{T_d} \models T(\phi_{T_d}^{\text{post}}(u, \vec{v}, \vec{z}, \vec{c}), \text{Return}_d) \) (assume the procedure call returns a location to \( u \)), where \( \vec{v} \) and \( \vec{z} \) are the actual parameters of the procedure call, \( \vec{c} \) are the compulsory variables with fresh names.

Now we are ready to define the verification condition corresponding to \( P \). We first derive a formula expressing that \( E \) does not involve null pointer dereference:

\[
\text{NullDereference} \equiv \bigwedge_{v \in \text{Deref}} v \neq \text{null}
\]

For each \( i \in [n] \), Figure 1 shows the effect of each statement on the verification condition generated. Each statement’s strongest post condition is captured in the logic, and for procedure calls, the heaplet manipulated by the procedure is carefully taken into account to update the heap at the caller. The conjunction of these

\[
\begin{align*}
\{ u := v \} & \quad \phi_i \equiv u = v \land R_i = R_{i-1} \land \text{FieldsUnmod}(PF \cup DF, i, i-1) \\
\{ u := \text{nil} \} & \quad \phi_i \equiv u = \text{nil} \land R_i = R_{i-1} \land \text{FieldsUnmod}(PF \cup DF, i, i-1) \\
\{ u := v.pf \} & \quad \\
\{ u.pf := v \} & \quad \phi_i \equiv u \in R_{i-1} \land u = pf_{T_{i-1}}(v) \land R_i = R_{i-1} \\
& \quad \land \text{FieldsUnmod}(PF \cup DF, i, i-1) \\
\{ j := u.df \} & \quad \\
\{ u.df := j \} & \quad \phi_i \equiv u \in R_{i-1} \land j = df_{T_{i-1}}(u) \land R_i = R_{i-1} \\
& \quad \land \text{FieldsUnmod}(PF \cup DF, i, i-1) \\
\{ j := \text{aexpr} \} & \quad \phi_i \equiv j = \text{aexpr} \land R_i = R_{i-1} \land \text{FieldsUnmod}(PF \cup DF, i, i-1) \\
\{ u := \text{new} \} & \quad \phi_i \equiv \text{new} \neq \text{null} \land u = \text{new} \land u \in R_{i-1} \land R_i = R_{i-1} \land |\text{new}| \\
& \quad \land \bigwedge_{i \in \text{pf}} (pf_i = pf_{T_{i-1}}(\text{new}) \land df_{T_{i-1}}(0 \leftarrow \text{new})) \\
\{ \text{free } u \} & \quad \phi_i \equiv u \in R_{T_{i-1}} \land R_i = R_{i-1} \land \text{FieldsUnmod}(PF \cup DF, i, i-1) \\
\{ \text{assume bexpr } \} & \quad \phi_i \equiv \text{bexpr} \land R_i = R_{i-1} \land \text{FieldsUnmod}(PF \cup DF, i, i-1) \\
\{ u := f(\vec{v}, \vec{z}) \} & \quad \phi_i \equiv T(\phi_{T_{i-1}}^{\text{pre}}(\vec{v}, \vec{z}, \vec{c}), \text{Call}_d)[i-1] \land T(\phi_{T_{i-1}}^{\text{post}}(u, \vec{v}, \vec{z}, \vec{c}), \text{Return}_d)[i] \\
& \quad \land (R_{T_{i-1}} \setminus \text{Call}_d) \cap \text{Return}_d = \emptyset \land R_i = (R_{T_{i-1}} \setminus \text{Call}_d) \cup \text{Return}_d \\
& \quad \text{where } d \text{ is the index such that } T_d = i \\
\{ j := g(\vec{v}, \vec{z}) \} & \quad \phi_i \text{ is defined in the same way as the above case,} \\
& \quad \text{except replacing } u \text{ with } j.
\end{align*}
\]

Figure 1. Formulas capture modification by statements.
formulas captures the modification made in $E$:

$$\text{Modification} \equiv \bigwedge_{i \in [n]} \psi_i$$

Finally, we can define two formulas to capture the pre- and post-conditions:

$$\text{Pre} \equiv T(\varphi_{pre}, R_0)[0]$$

$$\text{Post} \equiv T(\varphi_{post}, R_n)[n]$$

Now the validity of $(\varphi_{pre})P(\varphi_{post})$ can be captured by the following formula:

$$\psi_{VC} \equiv (\text{Pre} \land \text{NonNull Dereference} \land \text{Modification}) \rightarrow \text{Post}$$

B. Proof of Theorem 6.1

**Proof.** We prove the soundness by contradiction. Assume the Hoare-triple $[\varphi_{pre}]P[\varphi_{post}]$ is not valid. Assume $P$ consists of $n$ statements, then there is an execution $E$, which can be represented as a state sequence $(C_0, \ldots, C_n)$ where each $C_i = (R_i, s_i, h_i)$, such that $(C_0, R_0)$ satisfies $\psi_{pre}[0]$, $(C_n, R_n)$ satisfies $\psi_{post}[n]$, and the whole execution is memory error free. Then by the definitions of Pre, Post and NoNull Dereference and Theorem 6.1, $E \models \text{Pre} \land \text{NonNull Dereference} \land \text{Modification}$, and $E$ cannot be $\text{Post}$, in which case $E$ dissatisfies $\psi_{VC}$. The contradiction will complete the proof.

Since $\text{Modification} \equiv \bigwedge_{i \in [n]} \psi_i$, we just need to prove $E \models \psi_i$ for each $i \in [n]$, by case analysis on the type of the $i$-statement in $P$. 

1. $u := v$

   $\psi_i \equiv u = v \land R_i = R_{i-1} \land \text{FieldsUnmod}(P \land F \cup D, i, i-1)$

   The variable assignment makes $u$ points to where $v$ points to. Hence $u = v$. Since the heap is unmodified from $C_{i-1}$ to $C_i$, the heap domain remains the same ($R_i = R_{i-1}$), and all the field functions remain the same ($\text{FieldsUnmod}(P \land F \cup D, i, i-1)$).

2. $u := \text{null}$

   $\psi_i \equiv u = \text{null} \land R_i = R_{i-1} \land \text{FieldsUnmod}(P \land F \cup D, i, i-1)$

   The variable assignment makes $u$ points to null, so $u = \text{null}$. Similar to the above case, the heap is also unmodified from $C_{i-1}$ to $C_i$.

3. $u := v, pf$

   $\psi_i \equiv u \in R_{i-1} \land u = pf_{i-1}(v) \land R_i = R_{i-1}$

   The dereferencing on $v$ implies that $v$ points to a valid location at timestamp $i-1$, i.e., $v \in R_{i-1}$. Moreover, the assignment makes $u$ points to the $pf$ field of $v$ at timestamp $i-1$, formally $u = pf_{i-1}(v)$. Similar to the above case, the heap is also unmodified from $C_{i-1}$ to $C_i$.

4. $u, pf := v$

   $\psi_i \equiv u \in R_{i-1} \land pf_i = pf_{i-1}(v \leftarrow u) \land R_i = R_{i-1}$

   Similar to the above case, $u$ points to a valid location at timestamp $i-1$ ($u \in R_{i-1}$). Moreover, the assignment makes the $pf$ field at timestamp $i$ updated from that at timestamp $i-1$: $pf_i = pf_{i-1}(v \leftarrow u)$. Moreover, the heap domain is unmodified, so $R_i = R_{i-1}$. The other field functions also remain the same, which is captured by $\text{FieldsUnmod}(P \land (DF \setminus \{pf\}), i, i-1)$.

5. $u := u, df$

   $\psi_i \equiv u \in R_{i-1} \land j = df_{i-1}(u) \land R_i = R_{i-1}$

   Similar to the $u := v, pf$ case.

   $$\text{Post} \equiv T(\varphi_{post}, R_n)[n]$$

   $\equiv u \in R_{i-1} \land \bigwedge_{i \in [n]} \psi_i \land \text{FieldsUnmod}(P \land F \cup D, i, i-1)$$

   $\equiv u \in R_{i-1} \land u = v \land R_i = R_{i-1}$
C. Formulas Defined in Section 6.2

Let \( u \) be a location variable in \( LVars \) and let \( i \) be a timestamp such that \( 1 \leq i \leq n \). For each recursive definition \( \Delta \)-eliminated version defined as \( \text{rec}^\Delta(x) \equiv \text{def}^\Delta(x, \vec{i}, \vec{d}) \) and whose reach set defined as \( \text{reach}^\Delta(x) \equiv \text{reachdef}^\Delta(x) \), we can derive a formula \( \text{Unfold}^\Delta(i, u) \) for unfolding both \( \text{rec}^\Delta \) and its corresponding reach set on \( u \) at timestamp \( i \), provided that \( u \) is allocated at the current timestamp \( (u \in \mathcal{R}) \). Note that in \( \text{def}^\Delta(x, \vec{i}, \vec{d}) \), \( \vec{i} \) will be renamed as \( u \), and \( \vec{d} \) will not be renamed as they are program variables, but \( \vec{d} \) are existentially quantified and should be replaced with fresh variable names. Due to the restrictions on the recursive definitions, every \( e \) is unique and can be determined by dereferencing \( u \) on the corresponding pointer fields, say \( \text{pf}^\Delta \). Hence we can replace each \( v \) in \( \vec{d} \) distinctly as \( u \cdot \text{rec} \cdot \vec{d} \). Let the renamed formula be \( \text{def}^\Delta(u, \vec{i}, \vec{d}_{\text{fresh}}) \), then we can derive

\[
\text{Unfold}^\Delta(i, u) \equiv \left( \text{reach}^\Delta_i(u) = \text{reachdef}^\Delta_i(u) \right) \land \left( u \in \mathcal{R} \rightarrow \left( \text{rec}^\Delta(u) \equiv \text{def}^\Delta_i(u, \vec{i}, \vec{d}_{\text{fresh}}) \land \bigwedge_{e \in \vec{d}} \left( \text{pf}^\Delta_{\text{fresh}}(u) = u \cdot \text{rec} \cdot \vec{d} \right) \right) \right)
\]

Now the footprint unfolding is just unfolding \( u \) at the beginning and end of each program segment (for the \( d \)-th segment, the timestamp \( t_d \) and \( t_{d+1} - 1 \), respectively):

\[
\text{Unfold}^\Delta_d(u) \equiv \text{Unfold}^\Delta(i_d, u) \land \text{Unfold}^\Delta(i_{d+1} - 1, u)
\]

The formula \( \text{FieldUnchanged}^\Delta_d(u) \) describes that, in the \( d \)-th procedure call, if the location \( u \) is not \( \text{nil} \), then for each field \( \text{pf} \) (or \( \text{df} \)), \( \text{pf}_{i_d}(u) \) and \( \text{df}_{i_d}(u) \) are the same if \( u \) itself is not affected during the call:

\[
\text{FieldUnchanged}^\Delta_d(u) \equiv \left( u \neq \text{nil} \land u \notin \text{Call} \rightarrow \left( \bigwedge_{e \in \vec{d}} \left( \text{pf}_{i_d}(u) = \text{pf}_{i_d}(u) \land \bigwedge_{e \in \vec{d}} \left( \text{df}_{i_d}(u) = \text{df}_{i_d}(u) \right) \right) \right) \right)
\]

Finally, to define \( \text{RecUnchanged}^\Delta_d(u, i, i') \), we first define a formula expressing that a recursive definition and its corresponding reach set on a location are unchanged between two timestamps:

\[
\text{UnchangedBetween}^\Delta(u, i, i') \equiv \text{rec}^\Delta(u) = \text{rec}^\Delta(i) \land \text{reach}^\Delta(u) = \text{reach}^\Delta(i)
\]

For each non-footprint location variable \( u \) and for each recursive predicate \( \text{rec}^\Delta \), the formula \( \text{RecUnchanged}^\Delta(u, i, i') \) just captures the fact that \( \text{rec}^\Delta(u) \) and \( \text{reach}^\Delta(u) \) are unchanged in two cases: in the \( d \)-th segment of the program (between timestamp \( t_d \) and \( t_{d+1} - 1 \), they are unchanged if reach set is not modified; or in the \( d \)-th procedure call (between the timestamp \( t_d - 1 \) and \( t_d \), if the reach set is not affected during the call. Moreover, it also incorporates the fact that the reach set on \( u \) contains \( u \) itself. Formally,

\[
\text{RecUnchanged}^\Delta_d(u) \equiv \left( \text{reach}_{i_d}(u) \cap \text{Mod}_{i_d} = \emptyset \rightarrow \text{UnchangedBetween}^\Delta(u, t_d, t_{d+1} - 1) \right)
\land \left( \text{reach}_{i_d}(u) \cap \text{Call}_{i_d} = \emptyset \rightarrow \text{UnchangedBetween}^\Delta(u, t_d - 1, t_d) \right)
\land \left( u \neq \text{nil} \rightarrow \left( \text{reach}_{i_d}(u) \land \text{reach}^\Delta_{i_d+1}(u) \right) \right)
\]

D. Transforming \( \neg \psi_{\text{VC}}^{\text{abs}} \) to \( \psi_{\text{APF}} \)

Note that \( \neg \psi_{\text{VC}}^{\text{abs}} \) is mostly expressible in the quantifier-free theory of arrays, maps, uninterpreted functions, and integers: \( \text{Loc} \) can be viewed as an uninterpreted sort; each pointer field \( \text{pf} \) can be viewed as an array with both indices and elements of sort \( \text{Loc} \); each data field \( \text{df} \) can be viewed as an array with indices of sort \( \text{Loc} \) and elements of sort \( \text{Int} \); each integer set (or multiset) variable \( \text{S} \) can be viewed as an array with indices of sort \( \text{Int} \) and elements of sort \( \text{Bool} \) (or \( \text{Int} \)). Moreover, each array update operation of the form \( \text{array}[\text{elem} \leftarrow \text{key}] \) can be viewed as a read-over-write operation in the array property fragment, and each set-operation (union, intersection, etc.) can be viewed as a mapping function applying a Boolean operation \( \land, \lor, \text{etc} \) to the range of arrays.

The only construct in \( \neg \psi_{\text{VC}}^{\text{abs}} \) that escapes the quantifier-free formulation is the \( \leq \) relation between integer sets/multisets; but this can be captured using the array property fragment, which is decidable.

For each atomic formula of the form \( S_1 \leq S_2 \), if \( S_1 \) and \( S_2 \) are sets of integers, we can replace the formula with a universally quantified formula as follows:

\[
\forall i_1, i_2. \ ( i_1 \leq i_2 \rightarrow (S_2[i_1] \land \neg S_1[i_2])
\]

Similarly, if \( S_1 \) and \( S_2 \) are integer multisets, we can replace the formula with

\[
\forall i_1, i_2. \ ( i_1 \leq i_2 \rightarrow (S_2[i_1] \land \neg S_1[i_2]) = 0)
\]

The formula \( S_1 \leq S_2 \) where \( S_1 \) and \( S_2 \) are sets of integers can also be translated to

\[
\forall i. \ ( (S_1[i] \land i \leq k) \land (S_2[i] \land k \leq i))
\]

where \( k \) is an additional existential integer variable, serving as the pivot for splitting \( S_1 \) and \( S_2 \). Similarly, when \( S_1 \) and \( S_2 \) are integer multisets, the formula is translated to

\[
\forall i. \ ( (S_1[i] > 0 \land k \leq i) \land (S_2[i] > 0 \land k \leq i))
\]

Moreover, the negation of the above relations between sets/multisets can always be expressed using two existential integer variables \( k_1, k_2 \) that witness the violation of the inequality. For instance, \( S_1 \neq S_2 \) can be expressed as \( k_1 \in S_1 \land k_2 \in S_2 \land k_2 \leq k_1 \).

We thus obtain a formula \( \psi_{\text{APF}} \) whose satiability is decidable.