Natural Proofs for Structure, Data, and Separation

Supplemental Material

Xiaokang Qiu Pranav Garg Andrei Ştefănescu P. Madhusudan

University of Illinois at Urbana-Champaign, USA

{qiu2, garg11, stefane1, madhu}@illinois.edu

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):

$$S_0 \,;\, g_1 \,;\, S_1 \,;\, \ldots \,;\, g_m \,;\, S_m$$

where S_d is the $d+1$-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 $t_{-1} = t_0 = 0$ and $t_{m+1} = n+1$). Note that $\mathcal{E}$ 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 $C_{t_d-1} \mid Call_d$ and $C_{t_d} \mid Return_d$, respectively, where $Call_d \subseteq R_{t_d-1}$ and $Return_d \subseteq R_{t_d}$ are two sets of records.

Let all the location variables appearing in P be *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 $v.pf := u$ or $v.df := j$ in P. Then we can extract the set of dereferenced variables *Deref* and the set of modified variables *Mod*. Note that a modified variable is always dereferenced, i.e., $Mod \subseteq Deref$. For each basic segment S_d, let the dereferenced and modified variables within the segment be $Deref_{t_d}$ and Mod_{t_d}, respectively.

For the d-th procedure call, let the pre- and post-condition associated with the procedure be $\psi^d_{pre}(\vec{v}, \vec{z}, \vec{c})$ and $\psi^d_{post}(ret, \vec{v}, \vec{z}, \vec{c})$, respectively. Since $\mathcal{E}$ is a normal execution, we have $C_{t_d-1} \models T(\psi^d_{pre}(\vec{v_d}, \vec{z_d}, \vec{c_d}), Call_d)$ and $C_{t_d} \models T(\psi^d_{post}(u, \vec{v_d}, \vec{z_d}, \vec{c_d}), Return_d)$ (assume the procedure call returns a location to u), where $\vec{v_d}$ and $\vec{z_d}$ are the actual parameters of the procedure call, $\vec{c_d}$ are the complimentary variables with fresh names.

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

$$\textsc{NoNullDereference} \;\equiv\; \bigwedge_{v \in Deref} v \neq nil$$

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

[$u := v$]
$$\varphi_i \;\equiv\; u = v \wedge R_i = R_{i-1} \wedge FieldsUnmod(PF \cup DF, i, i-1)$$

[$u :=$ `nil`]
$$\varphi_i \;\equiv\; u = nil \wedge R_i = R_{i-1} \wedge FieldsUnmod(PF \cup DF, i, i-1)$$

[$u := v.pf$]
$$\begin{aligned}\varphi_i \;\equiv\;& v \in R_{i-1} \wedge u = pf_{i-1}(v) \wedge R_i = R_{i-1}\\ &\wedge FieldsUnmod(PF \cup DF, i, i-1)\end{aligned}$$

[$u.pf := v$]
$$\begin{aligned}\varphi_i \;\equiv\;& u \in R_{i-1} \wedge pf_i = pf_{i-1}\{v \leftarrow u\} \wedge R_i = R_{i-1}\\ &\wedge FieldsUnmod(PF \cup (DF \setminus \{pf\}), i, i-1)\end{aligned}$$

[$j := u.df$]
$$\begin{aligned}\varphi_i \;\equiv\;& u \in R_{i-1} \wedge j = df_{i-1}(u) \wedge R_i = R_{i-1}\\ &\wedge FieldsUnmod(PF \cup DF, i, i-1)\end{aligned}$$

[$u.df := j$]
$$\begin{aligned}\varphi_i \;\equiv\;& u \in R_{i-1} \wedge df_i = df_{i-1}\{j \leftarrow u\} \wedge R_i = R_{i-1}\\ &\wedge FieldsUnmod((PF \setminus \{df\}) \cup DF, i, i-1)\end{aligned}$$

[$j := aexpr$]
$$\varphi_i \;\equiv\; j = aexpr \wedge R_i = R_{i-1} \wedge FieldsUnmod(PF \cup DF, i, i-1)$$

[$u :=$ `new`]
$$\begin{aligned}\varphi_i \;\equiv\;& new_i \neq nil \wedge u = new_i \wedge new_i \notin R_{i-1} \wedge R_i = R_{i-1} \cup \{new_i\}\\ &\wedge \bigwedge_{pf} (pf_i = pf_{i-1}\{nil \leftarrow new_i\}) \wedge \bigwedge_{df} (df_i = df_{i-1}\{0 \leftarrow new_i\})\end{aligned}$$

[`free` u]
$$\varphi_i \;\equiv\; u \in R_{i-1} \wedge R_i = R_{i-1} \setminus \{u\} \wedge FieldsUnmod(PF \cup DF, i, i-1)$$

[`assume` $bexpr$]
$$\varphi_i \;\equiv\; bexpr \wedge R_i = R_{i-1} \wedge FieldsUnmod(PF \cup DF, i, i-1)$$

[$u := f(\vec{v}, \vec{z})$]
$$\begin{aligned}\varphi_i \;\equiv\;& T(\psi^d_{pre}(\vec{v}, \vec{z}, \vec{c_d}), Call_d)[i-1] \wedge T(\psi^d_{post}(u, \vec{v}, \vec{z}, \vec{c_d}), Return_d)[i]\\ &\wedge (R_{i-1} \setminus Call_d) \cap Return_d = \emptyset \wedge R_i = (R_{i-1} \setminus Call_d) \cup Return_d\\ &\text{where } d \text{ is the index such that } t_d = i\end{aligned}$$

[$j := g(\vec{v}, \vec{z})$]

φ_i is defined in the same way as the above case, except replacing u with j.

where $FieldsUnmod(F, i, j)$ is short for $\bigwedge_{field \in F}(field_i = field_j)$.

Figure 1. Formulas capture modification by statements

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'13, June 16–19, 2013, Seattle, WA, USA.

Copyright © 2013 ACM 978-1-4503-2014-6/13/06. . . $15.00

formulas captures the modification made in $\mathcal{E}$:

$$\textsc{Modification} \equiv \bigwedge_{i\in[n]} \varphi_i$$

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

$$\textsc{Pre} \equiv T(\psi_{pre}, R_0)[0]$$
$$\textsc{Post} \equiv T(\psi_{post}, R_n)[n]$$

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

$$\psi_{\mathrm{VC}} \equiv (\textsc{Pre} \wedge \textsc{NoNullDereference} \wedge \textsc{Modification}) \rightarrow \textsc{Post}$$

B. Proof of Theorem 6.1

Proof. We prove the soundness by contradiction. Assume the Hoare-triple $\{\psi_{pre}\}\ P\ \{\psi_{post}\}$ is not valid. Assume P consists of n statements, then there is an execution $\mathcal{E}$, 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 NoNullDereference, and Theorem 6.1, $\mathcal{E} \models \textsc{Pre} \wedge \textsc{NoNullDereference} \wedge \textsc{Post}$. Now it suffices to show that $\mathcal{E} \models \textsc{Modification}$, in which case $\mathcal{E}$ dissatisfies ψ_{VC}. The contradiction will conclude the proof.

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

[$u := v$]

$$\varphi_i \equiv u = v \wedge R_i = R_{i-1} \wedge \mathit{FieldsUnmod}(PF \cup DF, i, i-1)$$

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 ($\mathit{FieldsUnmod}(PF \cup DF, i, i-1)$).

[$u :=$ `nil`]

$$\varphi_i \equiv u = nil \wedge R_i = R_{i-1} \wedge \mathit{FieldsUnmod}(PF \cup DF, i, i-1)$$

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

[$u := v.pf$]

$$\varphi_i \equiv v \in R_{i-1} \wedge u = pf_{i-1}(v) \wedge R_i = R_{i-1} \wedge \mathit{FieldsUnmod}(PF \cup DF, i, i-1)$$

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.

[$u.pf := v$]

$$\varphi_i \equiv u \in R_{i-1} \wedge pf_i = pf_{i-1}\{v \leftarrow u\} \wedge R_i = R_{i-1} \wedge \mathit{FieldsUnmod}(PF \cup (DF \setminus \{pf\}), i, i-1)$$

Similar to the above case, u points to a valid location at timestamp $i-1$ ($u \in R_{i-1}$). the mutation 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 $\mathit{FieldsUnmod}(PF \cup (DF \setminus \{pf\}), i, i-1)$.

[$j := u.df$]

$$\varphi_i \equiv u \in R_{i-1} \wedge j = df_{i-1}(u) \wedge R_i = R_{i-1} \wedge \mathit{FieldsUnmod}(PF \cup DF, i, i-1)$$

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

[$u.df := j$]

$$\varphi_i \equiv u \in R_{i-1} \wedge df_i = df_{i-1}\{j \leftarrow u\} \wedge R_i = R_{i-1} \wedge \mathit{FieldsUnmod}((PF \setminus \{df\}) \cup DF, i, i-1)$$

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

[$j := aexpr$]

$$\varphi_i \equiv j = aexpr \wedge R_i = R_{i-1} \wedge \mathit{FieldsUnmod}(PF \cup DF, i, i-1)$$

The statement assigns the value of $aexpr$, which is expressible in our logic, to j. Hence $j = aexpr$. The rest is similar to other variable assignment cases.

[$u :=$ `new`]

$$\varphi_i \equiv new_i \neq nil \wedge u = new_i \wedge new_i \notin R_{i-1} \wedge R_i = R_{i-1} \cup \{new_i\} \wedge \bigwedge_{pf} (pf_i = pf_{i-1}\{nil \leftarrow new_i\}) \wedge \bigwedge_{df} (df_i = df_{i-1}\{0 \leftarrow new_i\})$$

This statement makes u points to a freshly allocated location, namely new_i in $\mathcal{E}$. So it is clear that $new_i \neq nil \wedge u = new_i$. Since the heap domain at timestamp i is an extension of that at timestamp $i-1$ by adding new_i, we know that $new_i \notin R_{i-1} \wedge R_i = R_{i-1} \cup \{new_i\}$. By default, for new_i, each pointer field initially points to nil, each data field initially stores 0. The remaining portion of the heap is exactly the same as C_{i-1}. Hence $\bigwedge_{pf}(pf_i = pf_{i-1}\{nil \leftarrow new_i\}) \wedge \bigwedge_{df}(df_i = df_{i-1}\{0 \leftarrow new_i\})$.

[`free` u]

$$\varphi_i \equiv u \in R_{i-1} \wedge R_i = R_{i-1} \setminus \{u\} \wedge \mathit{FieldsUnmod}(PF \cup DF, i, i-1)$$

This statement removes the location pointed by u from the heap. So the old heap contains this location, and the new heap can be obtained by subtracting it from the old heap: $u \in R_{i-1} \wedge R_i = R_{i-1} \setminus \{u\}$. Since the domain is shrinked, the field function can be simply unchanged.

[`assume` $bexpr$]

$$\varphi_i \equiv bexpr \wedge R_i = R_{i-1} \wedge \mathit{FieldsUnmod}(PF \cup DF, i, i-1)$$

The assumed condition $bexpr$, which can be expressed in our logic, must be true. The heap is simply unmodified.

[$u := f(\vec{v}, \vec{z})$]

$$\varphi_i \equiv T(\psi^d_{pre}(\vec{v}, \vec{z}, \vec{c_d}), Call_d)[i-1] \wedge T(\psi^d_{post}(u, \vec{v}, \vec{z}, \vec{c_d}), Return_d)[i] \wedge (R_{i-1} \setminus Call_d) \cap Return_d = \emptyset \wedge R_i = (R_{i-1} \setminus Call_d) \cup Return_d$$
$$\text{where } d \text{ is the index such that } t_d = i$$

As assumed, this call is the d-th procedure call in P, and C_{i-1} satisfies the associated precondition by the heaplet defined by $Call_d$, and C_i satisfies the associated postcondition by the heaplet defined by $Return_d$. Then formally we have $T(\psi^d_{pre}(\vec{v}, \vec{z}, \vec{c_d}), Call_d)[i-1] \wedge T(\psi^d_{post}(u, \vec{v}, \vec{z}, \vec{c_d}), Return_d)[i]$ being satisfied by $\mathcal{E}$.
Due to the framing property of the separation semantics, the portion of C_{i-1} that is not required by ψ_{pre} remains unchanged, and is disjoint from $Return_d$ (since the returned location is assigned to u, the variable ret can be replaced with u). This property can be expressed as $(R_{i-1} \setminus Call_d) \cap Return_d = \emptyset \wedge R_i = (R_{i-1} \setminus Call_d) \cup Return_d$.

[$j := g(\vec{v}, \vec{z})$]

φ_i is defined in the same way as the above case, except replacing u with j.

The proof is also similar to the above case. □

C. Formulas Defined in Section 6.2

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

$$\textsc{UnfoldAt}^{rec}(i, u) \equiv \Big(reach_i^{rec}(u) = \mathrm{reachdef}_i^{\,rec}(u) \Big) \wedge \Big(u \in R_i \rightarrow \Big(\big(rec_i(u) \leftrightarrow \mathrm{def}_i^{rec}(u, \vec{t}, \vec{v}_{fresh}) \big) \wedge \bigwedge_{v \in \vec{v}} \big(pf_i^{\,rec,v}(u) = u\_rec\_v\_i \big) \Big) \Big)$$

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):

$$\textsc{Unfold}_d^{rec}(u) \equiv \textsc{UnfoldAt}^{rec}(t_d, u) \wedge \textsc{UnfoldAt}^{rec}(t_{d+1} - 1, u)$$

The formula $\textsc{FieldUnchanged}_d(u)$ describes that, in the d-th procedure call, if the location u is not *nil*, then for each field *pf* (or *df*), $pf_{t_d - 1}(u)$ and $pf_{t_d}(u)$ are the same if u itself is not affected during the call:

$$\textsc{FieldUnchanged}_d(u) \equiv \Big((u \neq nil \wedge u \notin Call_d) \rightarrow \Big(\bigwedge_{pf} (pf_{t_d - 1}(u) = pf_{t_d}(u)) \wedge \bigwedge_{df} (df_{t_d - 1}(u) = df_{t_d}(u)) \Big) \Big)$$

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

$$\textsc{UnchangedBetween}^{rec}(u, i, i') \equiv rec_i(u) = rec_{i'}(u) \wedge reach_i^{rec}(u) = reach_{i'}^{rec}(u)$$

For each non-footprint location variable u and for each recursive predicate rec^{Δ}, the formula $\textsc{RecUnchanged}_d$ just captures the fact that $rec(u)$ and $reach^{rec}(u)$ 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,

$$\begin{aligned} &\textsc{RecUnchanged}_d^{rec}(u) \equiv \\ &\quad \Big((reach_{t_d}^{rec}(u) \cap Mod_{t_d} = \emptyset) \rightarrow \textsc{UnchangedBetween}^{rec}(u, t_d, t_{d+1} - 1) \Big) \\ &\quad \wedge \Big((reach_{t_d}^{rec}(u) \cap Call_d = \emptyset) \rightarrow \textsc{UnchangedBetween}^{rec}(u, t_d - 1, t_d) \Big) \\ &\quad \wedge \Big(u \neq nil \rightarrow (reach_{t_d}^{rec}(u) \cap reach_{t_{d+1}-1}^{rec}(u)) \Big) \end{aligned}$$

D. Transforming $\neg\psi_{\mathbf{VC}}^{\mathbf{abs}}$ to ψ^{APF}

Note that $\neg\psi_{\mathrm{VC}}^{\mathrm{abs}}$ is mostly expressible in the quantifier-free theory of arrays, maps, uninterpreted functions, and integers: *Loc* can be viewed as an uninterpreted sort; each pointer field *pf* can be viewed as an array with both indices and elements of sort *Loc*; each data field *df* can be viewed as an array with indices of sort *Loc* and elements of sort *Int*; each integer set (or multiset) variable S can be viewed as an array with indices of sort *Int* and elements of sort *Bool* (or *Int*). Moreover, each array update operation of the form $array\{elem \leftarrow key\}$ 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 ($\wedge$, $\vee$, etc.) to the range of arrays.

The only construct in $\neg\psi_{\mathrm{VC}}^{\mathrm{abs}}$ 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 $S_1 < S_2$, if S_1 and S_2 are sets of integers, we can be replace the formula with a universally quantified formula as follows:

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

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

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

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

$$\forall i.\ (\ (S_1[i] \rightarrow i \leq k) \wedge (S_2[i] \rightarrow k \leq i)\ )$$

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

$$\forall i.\ (\ (S_1[i] > 0 \rightarrow i \leq k) \wedge (S_2[i] > 0 \rightarrow k \leq i)\ )$$

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, $S_1 \not\leq S_2$ can be expressed as $k_1 \in S_1 \wedge k_2 \in S_2 \wedge k_2 \leq k_1$.

We thus obtain a formula ψ^{APF} whose satisfiability is decidable.