Lazy Abstraction & Spatial Interpolant

Reporter: Xie Li

December 17, 2021

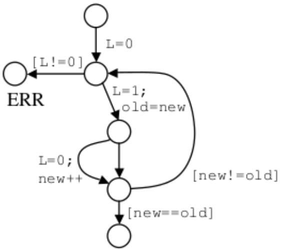
Papers

- Lazy Abstraction with Interpolants. CAV'06
- Spatial Interpolants. ESOP'15

Example: Lazy Abstraction with Interpolants

```
do{
  lock();
  old = new;
  if(*){
    unlock;
    new++;
  }
} while (new != old);
```

(a) program fragment



(b) control-flow graph

L is the variable for lock, locked if $\mathsf{L} = 1$.

Prove: L is always 0 on entry to `lock`.

- Unwind the program into a tree, and label the true statements for each tree node.
- Find a labeling that root is True and error states are False.

Interpolants

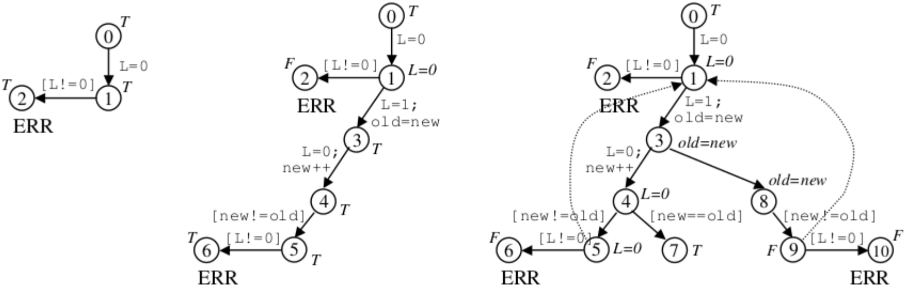
An interpolant for (A, B) is a formula A' s.t.

- $A \to A'$
- $A' \wedge B$ is unsatisfiable and
- $A' \in \mathcal{L}(A) \cap \mathcal{L}(B)$

We say $A'_0, \dots, A'_n$ is an interpolant for $\Gamma = A_1, \dots, A_n$ when

- $A'_0 =$ True, $A'_n =$ False
- For all $1 \le i \le n, A'_{i-1} \wedge A_i \implies A'_i$
- For all $1 \le i < n, A'_i \in \mathcal{L}(A_1 \dots A_i) \cap \mathcal{L}(A_{i+1} \dots A_n)$

Example: Unwinding



(a) first error (b) second error (c) termination

Dashed arrow represents the covering relation $\rhd$. e.g. $(5, 1) \in \rhd$.

Formalization: Program

- A program is a tuple $(\Gamma, \Delta, l_i, l_f)$. e.g. $(l, T, m) \in \Delta$, and T is a transition formula.
- A path π: $(l_0, T_0, l_1), (l_1, T_1, l_2), \ldots, (l_{n-1}, T_{n-1}, l_n)$. Error path.
- The unfolding $\mathcal{U}(\pi)$ of a path: $T_0^{\langle 0 \rangle}, \ldots, T_n^{\langle n-1 \rangle}$
- Inductive invariant: $I : \Lambda \to \mathcal{L}(S)$, s.t. $I(l_i) =$ True and for every $(l, T, m) \in \Delta$, $I(l) \wedge T \to I(m)'$.
- Safety invariant: An inductive invariant I s.t. $I(l_f) =$ False.

Formalization: Unwinding

Program unwindings We now give a definition of a program unwinding, and an algorithm to construct a complete unwinding using interpolants. For two vertices v and w of a tree, we will write $w \sqsubset v$ when w is a proper ancestor of v.

Definition 1. *An* unwinding *of a program* $\mathcal{A} = (\Lambda, \Delta, l_i, l_f)$ *is a quadruple* (V, E, M_v, M_e), *where* (V, E) *is a directed tree rooted at* ϵ, $M_v : V \to \Lambda$ *is the vertex map, and* $M_e : E \to \Delta$ *is the edge map, such that:*

- $M_v(\epsilon) = l_i$
- *for every non-leaf vertex* $v \in V$, *for every action* $(M_v(v), T, m) \in \Delta$, *there exists an edge* $(v, w) \in E$ *such that* $M_v(w) = m$ *and* $M_e(v, w) = T$.

Formalization: Unwinding

Definition 2. *A* labeled unwinding *of a program* $\mathcal{A} = (\Lambda, \Delta, l_i, l_f)$ *is a triple* $(U, \psi, \triangleright)$, *where*

- $U = (V, E, M_v, M_e)$ *is an unwinding of* $\mathcal{A}$
- $\psi : V \to \mathcal{L}(S)$ *is called the vertex labeling, and*
- $\triangleright \subseteq V \times V$ *is called the covering relation.*

A vertex $v \in V$ *is said to be* covered *iff there exists* $(w, x) \in \triangleright$ *such that* $w \sqsubseteq v$. *The unwinding is said to be* safe *iff, for all* $v \in V$, $M_v(v) = l_f$ *implies* $\psi(v) \equiv$ FALSE. *It is* complete *iff every leaf* $v \in V$ *is covered.*

Definition 3. *A* labeled unwinding $(U, \psi, \triangleright)$ *of a program* $\mathcal{A} = (\Lambda, \Delta, l_i, l_f)$, *where* $U = (V, E, M_v, M_e)$, *is said to be* well-labeled *iff:*

- $\psi(\epsilon) \equiv$ TRUE, *and*
- *for every edge* $(v, w) \in E$, $\psi(v) \wedge M_e(v, w)$ *implies* $\psi(w)'$, *and*
- *for all* $(v, w) \in \triangleright$, $\psi(v) \Rightarrow \psi(w)$, *and* w *is not covered.*

Soundness

Theorem (Soundness)

If there exists a safe, complete, well-labeled unwinding of program $\mathcal{A}$, then $\mathcal{A}$ is safe.

Proof.

Let U be the set of uncovered vertices, and let function M map location l to $\bigvee\{\psi(v) \mid M_v(v) = l, v \in U\}$. M is a safety invariant for $\mathcal{A}$. $\square$

Algorithm

global variables: V a set, $E \subseteq V \times V$, $\triangleright \subseteq V \times V$ and $\psi : V \rightarrow \mathit{wff}$

procedure $\textsc{Expand}(v \in V)$:
  if v is an uncovered leaf then
    for all actions $(M_v(v), T, m) \in \Delta$
      add a new vertex w to V and a new edge (v, w) to E;
      set $M_v(w) \leftarrow m$ and $\psi(w) \leftarrow \textsc{True}$;
      set $M_e(v, w) \leftarrow T$

procedure $\textsc{Refine}(v \in V)$:
  if $M_v(v) = l_f$ and $\psi(v) \not\equiv \textsc{False}$ then
    let $\pi = (v_0, T_0, v_1) \cdots (v_{n-1}, T_{n-1}, v_n)$ be the unique path from ϵ to v
    if $\mathcal{U}(\pi)$ has an interpolant $\hat{A}_0, \ldots, \hat{A}_n$ then
      for $i = 0 \ldots n$:
        let $\phi = \hat{A}_i^{\langle -i \rangle}$
        if $\psi(v_i) \not\models \phi$ then
          remove all pairs $(\cdot, v_i)$ from $\triangleright$
          set $\psi(v_i) \leftarrow \psi(v_i) \wedge \phi$
    else abort (program is unsafe)

procedure $\textsc{Cover}(v, w \in V)$:
  if v is uncovered and $M_v(v) = M_v(w)$ and $v \not\sqsubseteq w$ then
    if $\psi(v) \models \psi(w)$ then
      add (v, w) to $\triangleright$;
      delete all $(x, y) \in \triangleright$, s.t. $v \sqsubseteq y$;

Algorithm

procedure $\textsc{Close}(v \in V)$:
    for all $w \in V$ s.t. $w \prec v$ and $M_v(w) = M_v(v)$:
        $\textsc{Cover}(v, w)$

recursive procedure $\mathrm{DFS}(v \in V)$:
    $\textsc{Close}(v)$
    if v is uncovered then
        if $M_v(v) = l_f$ then
            $\textsc{Refine}(v)$;
            for all $w \sqsubseteq v$: $\textsc{Close}(w)$
        $\textsc{Expand}(v)$;
        for all children w of v: $\mathrm{DFS}(w)$

procedure $\textsc{Unwind}$:
    set $V \leftarrow \{\epsilon\}$, $E \leftarrow \emptyset$, $\psi(\epsilon) \leftarrow \textsc{True}$, $\triangleright \leftarrow \emptyset$
    while there exists an uncovered leaf $v \in V$:
        for all $w \in V$ s.t. $w \sqsubset v$: $\textsc{Close}(w)$;
        $\mathrm{DFS}(v)$

where $\prec$ is the relation regarding $\sqsubseteq$, indicating the order of computing the covering relation. Since adding a new pair may delete old pairs.

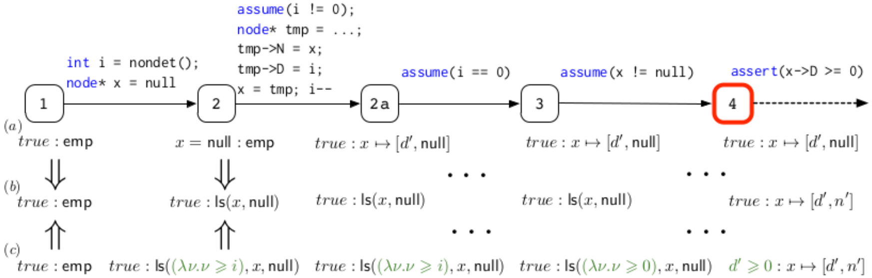
Lazy Abstraction with Spatial Interpolants

Procedure:

- Sample a program path and construct a Hoare stype proof of this path.
- Compute Spatial Interpolants
- Refine with Theory Interpolants
- From Proofs of Paths to Proofs of Programs

Example: Spatial Interpolants

```
1: int i = nondet();
   node* x = null;
2: while (i != 0)
     node* tmp = malloc(node);
     tmp->N = x;
     tmp->D = i;
     x = tmp;
     i--;
3: while (x != null)
4:   assert(x->D >= 0);
     x = x->N;
```



Preliminaries: Separation Logic

$$
\begin{array}{lll||rcl}
x, y \in \mathsf{HVar} & & \text{(Heap variables)} & E, F \in \mathsf{HTerm} & ::= & \mathsf{null} \mid x \\
a, b \in \mathsf{DVar} & & \text{(Data variables)} & \text{Æ} & ::= & A \mid E \\
A \in \mathsf{DTerm} & & \text{(Data terms)} & \Pi \in \mathsf{Pure} & ::= & \mathit{true} \mid E = E \mid E \neq E \mid \\
\varphi \in \mathsf{DFormula} & & \text{(Data formulas)} & & & \varphi \mid \Pi \wedge \Pi \\
Z \in \mathsf{RPred} & & \text{(Rec. predicates)} & H \in \mathsf{Heaplet} & ::= & \mathsf{true} \mid \mathsf{emp} \mid E \mapsto [\vec{A}, \vec{E}] \mid Z(\vec{\theta}, \vec{E}) \\
\theta \in \mathsf{Refinement} & ::= & \lambda \vec{a}.\varphi & \Sigma \in \mathsf{Spatial} & ::= & H \mid H * \Sigma \\
X \subseteq \mathsf{Var} & ::= & x \mid a & P \in \mathsf{RSep} & ::= & (\exists X.\ \Pi : \Sigma)
\end{array}
$$

Fig. 4. Syntax of RSep formulas.

Features:

- General recursive predicates.
- Recursive predicates are augmented with a vector of refinements on data values.
- Each heal cell is a record consisting of data fields followed by heap fields.
- Pure formulas contain heal and first-order data constraints.

Preliminaries: Separation Logic

Separation conjunction:

$$P * Q = (\exists X_P \cup X_Q . \Pi_P \wedge \Pi_Q : \Sigma_P * \Sigma_Q)$$

We write $\exists X . P$ to denote:

$$\exists X . P = \exists X \cup X_P . \Pi_P : \Sigma_P$$

Recursive predicates:

$$\begin{aligned}
\mathsf{ls}(R, x, y) \equiv\ & (x = y : \mathsf{emp}) \vee \\
& (\exists d, n'.\ x \neq y \wedge R(d) : x \mapsto [d, n'] * \mathsf{ls}(R, n', y)) \\
\mathsf{bt}(Q, L, R, x) =\ & (x = \mathsf{null} : \mathsf{emp}) \\
& \vee (\exists d, l, r.\ Q(d) : x \mapsto [d, l, r] \\
& \quad * \mathsf{bt}((\lambda a. Q(a) \wedge L(d, a)), L, R, l) \\
& \quad * \mathsf{bt}((\lambda a. Q(a) \wedge R(d, a)), L, R, r))
\end{aligned}$$

$$\mathsf{bt}((\lambda a. true), (\lambda a, b. a \geqslant b), (\lambda a, b. a \leqslant b), x)$$

Prelimiaries: Separation Logic

Generally

$$Z(\vec{R}, \vec{x}) \equiv (\exists X_1.\ \Pi_1 \wedge \Phi_1 : \Sigma_1) \vee \cdots \vee (\exists X_n.\ \Pi_n \wedge \Phi_n : \Sigma_n)$$

Semantic of recursive predicates:

$$s, h \models Z(\vec{\theta}, \vec{E}) \iff \exists P \in cases(Z(\vec{R}, \vec{x})).\ s, h \models P[\vec{\theta}/\vec{R}, \vec{E}/\vec{x}]$$

Prelimiaries: Program

Program is a tuple $\langle V, E, v_i, v_e \rangle$.
Each edge $e \in E$ is connected with an edge command e^c.

Assignment: x := Æ
Heap store: x->N$_i$:= E
Heap load: y := x->N$_i$
Assumption: assume(Π)
Data store: x->D$_i$:= A
Data load: y := x->D$_i$
Allocation: x := new(n, m)
Disposal: free(x)

Phase 1: Forward Symbolic Execution

For heap statements:

$$\texttt{exec(x := new}(k,l)\texttt{)},\ (\exists X.\ \Pi : \Sigma)) = (\exists X \cup \{x', \vec{d}, \vec{n}\}.\ (\Pi : \Sigma)[x'/x] * x \mapsto [\vec{d}, \vec{n}])$$
where $x', \vec{d}, \vec{n}$ are fresh, $\vec{d} = (d_1, \ldots, d_k)$, and $\vec{n} = (n_1, \ldots, n_l)$.

$$\texttt{exec(free(x)},\ (\exists X.\ \Pi : \Sigma * z \mapsto [\vec{d}, \vec{n}]) = (\exists X.\ \Pi \wedge \Pi^{\neq} : \Sigma)$$
where $\Pi : \Sigma * z \mapsto [\vec{d}, \vec{n}] \vdash x = z$ and $\Pi^{\neq}$ is the conjunction of all disequalities $x \neq y$ s.t $y \mapsto [\_, \_] \in \Sigma$.

$$\texttt{exec(x := E},\ (\exists X.\ \Pi : \Sigma)) = (\exists X \cup \{x'\}.\ (x = E[x'/x]) * (\Pi : \Sigma)[x'/x])$$
where x' is fresh.

$$\texttt{exec(assume}(\Pi')\texttt{)},\ (\exists X.\ \Pi : \Sigma)) = (\exists X.\ \Pi \wedge \underline{\Pi'} : \Sigma)\ .$$

$$\texttt{exec(x->N}_i\texttt{ := E},\ (\exists X.\ \Pi : \Sigma * z \mapsto [\vec{d}, \vec{n}])) = (\exists X.\ \Pi : \Sigma * x \mapsto [\vec{d}, \vec{n}[E/n_i]])$$
where $i \leqslant |\vec{n}|$ and $\Pi : \Sigma * z \mapsto [\vec{d}, \vec{n}] \vdash x = z$.

$$\texttt{exec(y := x->N}_i,\ (\exists X.\ \Pi : \Sigma * z \mapsto [\vec{d}, \vec{n}])) =$$
$$(\exists X \cup \{y'\}.\ (y = n_i[y'/y]) * (\Pi : \Sigma * z \mapsto [\vec{d}, \vec{n}])[y'/y])$$
where $i \leqslant |\vec{n}|$ and $\Pi : \Sigma * z \mapsto [\vec{d}, \vec{n}] \vdash x = z$, and y' is fresh.

Phase 2: Backward Interpolation Phase

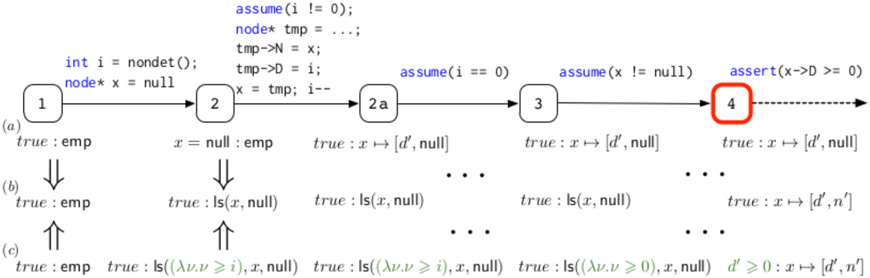
Spatial Interpolants:

Definition 1 (Spatial path interpolant). *Let $\pi = e_1, \ldots, e_n$ be a program path with symbolic execution sequence $S_0, \ldots, S_n$, and let P be a* Sep *formula (such that $S_n \models P$). A* spatial path interpolant *for π is a sequence $I_0, \ldots, I_n$ of* Sep *formulas such that*

- *for each $i \in [0, n]$, $S_i \models I_i$;*
- *for each $i \in [1, n]$, $\{I_{i-1}\}\ e_i^c\ \{I_i\}$ is a valid triple in separation logic; and*
- *$I_n \models P$.*

Definition 2 (Spatial interpolant). *Given* Sep *formulas S and I' and a command* c *such that* exec(c, S) $\models I'$, *a* spatial interpolant *(for S,* c*, and I') is a* Sep *formula I such that $S \models I$ and $\{I\}$* c *$\{I'\}$ is valid.*

Example Review



Definition 2 (Spatial interpolant). *Given Sep formulas S and I' and a command* c *such that* $\mathsf{exec}(\mathsf{c}, S) \models I'$, *a* spatial interpolant *(for S, c, and I') is a Sep formula I such that $S \models I$ and $\{I\}\ \mathsf{c}\ \{I'\}$ is valid.*

Bounded Abduction:

Definition 3 (Bounded abduction). *Let L, M, R be* Sep *formulas, and let X be a set of variables. A solution to the* bounded abduction problem

$$L \vdash (\exists X.\ M * [\ ]) \vdash R$$

is a Sep *formula A such that $L \models (\exists X.\ M * A) \models R$.*

Bounded Abduction Rules

Allocate Suppose `c` is `x := new(`n, m`)`. We take $\mathsf{itp}(S, \mathsf{c}, I') = (\exists x.\ A)$, where A is obtained as a solution to $\mathsf{exec}(\mathsf{c}, S) \vdash (\exists \vec{a}, \vec{z}.\ x \mapsto [\vec{a}, \vec{z}] * [A]) \vdash I'$, and $\vec{a}$ and $\vec{z}$ are vectors of fresh variables of length n and m, respectively.

Deallocate Suppose `c` is `free(x)`. We take $\mathsf{itp}(S, \mathsf{c}, I') = (\exists \vec{a}, \vec{z}.\ I' * x \mapsto [\vec{a}, \vec{z}])$, where $\vec{a}$ and $\vec{z}$ are vectors of fresh variables whose lengths are determined by the unique heap cell which is allocated to x in S.

Assignment Suppose `c` is `x := E`. We take $\mathsf{itp}(S, \mathsf{c}, I') = I'[E/x]$.

Bounded Abduction Rules for Load and Store

Store Suppose `c` is `x->N`$_i$ `:= E`. We take $\mathsf{itp}(S, \mathsf{c}, I') = (\exists \vec{a}, \vec{z}.\ A * x \mapsto [\vec{a}, \vec{z}])$, where A is obtained as a solution to $\mathsf{exec}(\mathsf{c}, S) \vdash (\exists \vec{a}, \vec{z}.\ x \mapsto [\vec{a}, \vec{z}[E/z_i]] * [A]) \vdash I'$ and where $\vec{a}$ and $\vec{z}$ are vectors of fresh variables whose lengths are determined by the unique heap cell which is allocated to x in S.

Load Suppose `c` is `y := x->N`$_i$. Suppose that $\vec{a}$ and $\vec{z}$ are vectors of fresh variables of lengths $|\vec{A}|$ and $|\vec{E}|$ where S is of the form $\Pi : \Sigma * w \mapsto [\vec{A}, \vec{E}]$ and $\Pi : \Sigma * w \mapsto [\vec{A}, \vec{E}] \vdash x = w$ (this is the condition under which $\mathsf{exec}(\mathsf{c}, S)$ is defined, see Fig. 5). Let y' be a fresh variable, and define $\overline{S} = (y = z_i[y'/y]) * (\Pi : \Sigma * w \mapsto [\vec{a}, \vec{z}])[y'/y]$. Note that $\overline{S} \vdash (\exists y'.\ \overline{S}) \equiv \mathsf{exec}(\mathsf{c}, S) \vdash I'$.

We take $\mathsf{itp}(S, \mathsf{c}, I') = (\exists \vec{a}, \vec{z}.\ A[z_i/y, y/y'] * x \mapsto [\vec{a}, \vec{z}])$ where A is obtained as a solution to $\overline{S} \vdash (\exists \vec{a}, \vec{z}.\ x[y'/y] \mapsto [\vec{a}, \vec{z}] * [A]) \vdash I'$.

Bounded Abduction Rules for Assume

Example:

$$\texttt{itp}(S, \texttt{assume(x != null)}, (\exists a, z. x \mapsto [a, z] * \texttt{true}))$$

For c is `assume(E != F)`

to introduce recursive predicates for the `assume` interpolation rules. Let P, Q be Sep formulas such that $P \vdash Q$, let Z be a recursive predicate and $\vec{E}$ be a vector of heap terms. We define $\mathsf{intro}(Z, \vec{E}, P, Q)$ as follows: if $P \vdash (\exists \emptyset.\ Z(\vec{E}) * [A]) \vdash Q$ has a solution and $A \nvdash Q$, define $\mathsf{intro}(Z, \vec{E}, P, Q) = Z(\vec{E}) * A$. Otherwise, define $\mathsf{intro}(Z, \vec{E}, P, Q) = Q$.

At last we take $\texttt{itp}(S, \mathsf{c}, I')$ to be M, where

$$M = \mathsf{intro}(Z_1, \vec{E}_1, S \wedge E \neq F, \mathsf{intro}(Z_2, \vec{E}_2, S \wedge E \neq F, \dots))$$

How is Bounded Abduction Conducted

Problem:

$$L \vdash \exists.M * [A] \vdash R$$

High level description:

- Find a coloring of L, assign color red or blue to the heaplets in L. Red for satifying M, blues are left overs.
- Compute the color strengthening $[M'] * [A]$ for R by recursion on the proof of $L \vdash R$.
- Check $M * A \models R$, if failed the algorithm fails to compute the solution for the problem.

Phase 3: Spatial Interpolation Modulo Theories

Refined memory safety proof ζ'	Constraint system $\mathcal{C}$	Solution σ
$\{R_0(i) : \mathsf{true}\}$	$R_0(i') \leftarrow true$	$R_0(i) : true$
`i = nondet(); x = null`	$R_1(i') \leftarrow R_0(i)$	$R_1(i) : true$
$\{R_1(i) : \mathsf{ls}((\lambda a.R_{\mathsf{ls}1}(\nu, i)), x, \mathsf{null}) * \mathsf{true}\}$	$R_2(i') \leftarrow R_1(i) \wedge i \neq 0 \wedge i' = i + 1$	$R_2(i) : true$
`assume(i != 0); ...; i--;`	$R_3(i) \leftarrow R_2(i) \wedge i = 0$	$R_3(i) : true$
$\{R_2(i) : \mathsf{ls}((\lambda a.R_{\mathsf{ls}2}(\nu, i)), x, \mathsf{null}) * \mathsf{true}\}$	$R_4(i, d') \leftarrow R_3(i) \wedge R_{\mathsf{ls}3}(d', i)$	$R_4(i, d') : d' \geqslant 0$
`assume(i == 0)`	$R_{\mathsf{ls}2}(\nu, i') \leftarrow R_1(i) \wedge R_{\mathsf{ls}1}(\nu, i) \wedge i \neq 0 \wedge i' = i + 1$	$R_{\mathsf{ls}1}(\nu, i) : \nu \geqslant i$
$\{R_3(i) : \mathsf{ls}((\lambda a.R_{\mathsf{ls}3}(\nu, i)), x, \mathsf{null}) * \mathsf{true}\}$	$R_{\mathsf{ls}2}(\nu, i') \leftarrow R_1(i) \wedge \nu = i \wedge i \neq 0 \wedge i' = i + 1$	$R_{\mathsf{ls}2}(\nu, i) : \nu \geqslant i$
`assume(x != null)`	$R_{\mathsf{ls}3}(\nu, i) \leftarrow R_2(i) \wedge R_{\mathsf{ls}2}(\nu, i) \wedge i = 0$	$R_{\mathsf{ls}3}(\nu, i) : \nu \geqslant 0$
$\{(\exists d', y.\ R_4(i, d') : x \mapsto [d', y] * \mathsf{true})\}$	$d' \geqslant 0 \leftarrow R_4(i, d')$	

Fig. 7. Example constraints.