# Remarks on Preconditon Generation (Footprints, Abduction, etc)

Peter O'Hearn
SNU, 12 May 2009

Based on joint work with Cristiano Calcagno, Dino Distefano, Hongseok Yang

Recall from Yesterday how we used Abduction to generate specifications (preconditions) for bare code

# *Abduction Example: Inferring a pre/post pair*

| | | | |
|---|---|---|---|
| 1 | void p(list-item *y) { | emp | list($y$)(Inferred Pre) |
| 2 | list-item *x; | | |
| 3 | x=malloc(sizeof(list-item)); | | |
| 4 | x$\rightarrow$tail = 0; | $x \mapsto 0$ | |
| 5 | foo(x,y); | list($x$) | |
| 6 | return(x); } | | list($ret$)(Inferred Post) |

Abductive Inference: $x \mapsto 0 * \text{list}(y) \vdash \text{list}(x) * \text{list}(y)$

Given Summary/spec: $[\text{list}(x) * \text{list}(y)]\, foo(x, y)\, [\text{list}(x)]$

Today we will talk a bit more in depth about that, including some of the trickinesses.

This talk is intended to be informal. Fell free to ask questions (if I can't answer, I will refer you to...)

# *Example*

$\{\texttt{emp}\}$
$x{=}\texttt{nil};$
*while* $(\_\ )\{$
$\quad\quad\texttt{new}(y);$
$\quad\quad y\texttt{->}tl = x;$
$\quad\quad x{=}y;$
$\}$

Calculated Loop Invariant

$\vee$

$\vee$

# *Example*

$\{\texttt{emp}\}$
$x$=nil;
*while* (_ ){ $\quad x = \texttt{nil} \wedge \texttt{emp}$
new($y$);
$y$ ->$tl = x$;
$x$=$y$;
}

Calculated Loop Invariant

$$x = \texttt{nil} \wedge \texttt{emp}$$
$$\vee$$
$$\vee$$

# *Example*

$\{\texttt{emp}\}$
$x{=}\texttt{nil};$
*while* $(\_\,)\{$ $\quad x \mapsto \texttt{nil}$
$\quad \texttt{new}(y);$
$\quad y\texttt{->}tl = x;$
$\quad x{=}y;$
$\}$

Calculated Loop Invariant

$$\begin{array}{ll} & x = \texttt{nil} \wedge \texttt{emp} \\ \vee & x \mapsto \texttt{nil} \\ \vee & \end{array}$$

# *Example*

$\{\texttt{emp}\}$
$x{=}\texttt{nil};$
*while* $(\_\,)\{$ $\quad x \mapsto x' * x' \mapsto \texttt{nil}$
$\quad\quad \texttt{new}(y);$
$\quad\quad y\texttt{->}tl = x;$
$\quad\quad x{=}y;$
$\}$

Calculated Loop Invariant

$$\begin{array}{ll} & x = \texttt{nil} \wedge \texttt{emp} \\ \vee & x \mapsto \texttt{nil} \\ \vee & \end{array}$$

# *Example*

$\{\texttt{emp}\}$
$x{=}\texttt{nil};$
*while* $(\_\,)\{$ $\quad \mathsf{ls}(x, \texttt{nil})$
$\quad \texttt{new}(y);$
$\quad y\texttt{->}tl = x;$
$\quad x{=}y;$
$\}$

Calculated Loop Invariant

$$\begin{array}{ll} & x = \texttt{nil} \wedge \mathsf{emp} \\ \vee & x \mapsto \texttt{nil} \\ \vee & \mathsf{ls}(x, \texttt{nil}) \end{array}$$

# *Example*

$\{\texttt{emp}\}$
$x{=}\texttt{nil};$
*while* $(\_\,)\{$ $\qquad x \mapsto x' * \mathsf{ls}(x', \texttt{nil})$
$\quad \texttt{new}(y);$
$\quad y\texttt{->}tl = x;$
$\quad x{=}y;$
$\}$

Calculated Loop Invariant

$$\begin{array}{ll} & x = \texttt{nil} \wedge \mathsf{emp} \\ \vee & x \mapsto \texttt{nil} \\ \vee & \mathsf{ls}(x, \texttt{nil}) \end{array}$$

# *Example*

$\{\texttt{emp}\}$
$x{=}\texttt{nil};$
*while* $(\_)\{$ $\quad\quad \mathsf{ls}(x, \texttt{nil})$
$\quad$ $\texttt{new}(y);$
$\quad$ $y\texttt{->}tl = x;$
$\quad$ $x{=}y;$
$\}$

Calculated Loop Invariant

$$\begin{array}{ll} & x = \texttt{nil} \wedge \mathsf{emp} \\ \vee & x \mapsto \texttt{nil} \\ \vee & \mathsf{ls}(x, \texttt{nil}) \end{array}$$

# *Example*

$\{\texttt{emp}\}$
$x{=}\texttt{nil};$
*while* $(\_)\{$ $\quad \mathsf{ls}(x, \texttt{nil})$
$\quad$ `new`$(y);$
$\quad y\texttt{->}tl = x;$
$\quad x{=}y;$
$\}$

Calculated Loop Invariant

$$\begin{array}{ll} & x = \texttt{nil} \wedge \mathsf{emp} \\ \vee & x \mapsto \texttt{nil} \\ \vee & \mathsf{ls}(x, \texttt{nil}) \end{array}$$

Fixed-point reached!

# *Cooking a Program Analyzer*

1. Just write an interpreter. (Well, an *abstract* interpreter.)
2. Symbolically execute statements using in-place reasoning (all true Hoare triples).
3. Interpret while loops by using abstractin rules like

$$\mathsf{ls}(x, t') * \mathsf{list}(t') \vdash \mathsf{list}(x)$$

   to automatically find loop invariants. This uses the rule of consequence on the right to find the invariant for the while rule

$$\frac{\{P\}C\{Q\} \quad Q \vdash Q'}{\{P\}C\{Q'\}} \qquad \frac{\{I \wedge B\}C\{I\}}{\{I\}\texttt{while } B \texttt{ do } \{I \wedge \neg B\}}$$

4. A terminating run of the interpreter will give us a **proof** of assertions at all program points.

# Abstraction, fixed-points, and Preconditon Generation

```
1:  while (c!=NULL) {
2:      t=c;
3:      c=c->tl;
4:      free(t);
5:  }
```

Discovered Precondition: $\texttt{c==c\_} \wedge \text{lseg}(\texttt{c\_},\texttt{NULL})$

| | Current Heap | Footprint Heap |
|---|---|---|
| First iteration | | |
| **pre:** | c!=NULL ∧ c==c_ ∧ t==c_ ∧ emp | emp |
| **post:** | t!=NULL ∧ c==c1_ ∧ t==c_ ∧ c_ ↦ c1_ | c_ ↦ c1_ |
| Second Iteration | | |
| **pre:** | c!=NULL ∧ c==c1_ ∧ t==c1_ ∧ emp | c_ ↦ c1_ |
| **post:** | t!=NULL ∧ c==c2_ ∧ t==c1_ ∧ c1_ ↦ c2_ | c_ ↦ c1_ * c1_ ↦ c2_ |
| **abs post:** | t!=NULL ∧ c==c2_ ∧ t==c1_ ∧ c1_ ↦ c2_ | lseg(c_,c2_) |
| Third Iteration | | |
| **pre:** | c!=NULL ∧ c==c2_ ∧ t==c2_ ∧ emp | lseg(c_,c2_) |
| **post:** | t!=NULL ∧ c==c3_ ∧ t==c2_ ∧ c2_ ↦ c3_ | lseg(c_,c2_) * c2_ ↦ c3_ |
| **abs post:** | t!=NULL ∧ c==c3_ ∧ t==c2_ ∧ c2_ ↦ c3_ | lseg(c_,c3_) |

**Fig. 2.** Pre and Post States at line 3 during footprint analysis of `delete_list`

| | Current Heap | Footprint Heap |
|---|---|---|
| First iteration | | |
| **pre:** | c!=NULL ∧ c==c_ ∧ t==c_ ∧ emp | emp |
| **post:** | t!=NULL ∧ c==c1_ ∧ t==c_ ∧ c_ ↦ c1_ | c_ ↦ c1_ |
| Second Iteration | | |
| **pre:** | c!=NULL ∧ c==c1_ ∧ t==c1_ ∧ emp | c_ ↦ c1_ |
| **post:** | t!=NULL ∧ c==c2_ ∧ t==c1_ ∧ c1_ ↦ c2_ | c_ ↦ c1_ * c1_ ↦ c2_ |
| **abs post:** | t!=NULL ∧ c==c2_ ∧ t==c1_ ∧ c1_ ↦ c2_ | lseg(c_,c2_) |
| Third Iteration | | |
| **pre:** | c!=NULL ∧ c==c2_ ∧ t==c2_ ∧ emp | lseg(c_,c2_) |
| **post:** | t!=NULL ∧ c==c3_ ∧ t==c2_ ∧ c2_ ↦ c3_ | lseg(c_,c2_) * c2_ ↦ c3_ |
| **abs post:** | t!=NULL ∧ c==c3_ ∧ t==c2_ ∧ c2_ ↦ c3_ | lseg(c_,c3_) |

**Fig. 2.** Pre and Post States at line 3 during footprint analysis of `delete_list`

```
1:  while (c!=NULL) {
2:     t=c;
3:     c=c->tl;
4:     free(t);
5:  }
```

Discovered Precondition: c==c_ ∧ lseg(c_,NULL)

Finally, we can exit the loop by removing "`c2_` $\mapsto$ `c3_`" from the current heap in the `free` command, and adding the negation of the loop conditional to the heap and footprint, and forgetting about `t` because it is a local variable. A bit of logic tells us that the footprint part is equivalent to `c3_ == NULL` $\wedge$ `lseg(c_,c3_)`, and when we add this to the initial precondition `c== c_` we obtain the overall precondition pictured in Figure 1.

# *Cooking a Program Analyzer*

1. Just write an interpreter. (Well, an *abstract* interpreter.)
2. Symbolically execute statements using in-place reasoning (all true Hoare triples).
3. Interpret while loops by using abstractin rules like

$$\mathsf{ls}(x, t') * \mathsf{list}(t') \vdash \mathsf{list}(x)$$

   to automatically find loop invariants. This uses the rule of consequence on the right to find the invariant for the while rule

$$\frac{\{P\}C\{Q\} \quad Q \vdash Q'}{\{P\}C\{Q'\}} \qquad \frac{\{I \wedge B\}C\{I\}}{\{I\}\texttt{while } B \texttt{ do } \{I \wedge \neg B\}}$$

4. A terminating run of the interpreter will give us a **proof** of assertions at all program points.

# *Cooking a Program Analyzer*

1. Just write an interpreter. (Well, an *abstract* interpreter.)
2. Symbolically execute statements using in-place reasoning (all true Hoare triples).
3. Interpret while loops by using abstractin rules like

$$\mathsf{ls}(x, t') * \mathsf{list}(t') \vdash \mathsf{list}(x)$$

   to automatically find loop invariants. This uses the rule of consequence on the right to find the invariant for the while rule

$$\frac{\text{P|-P'} \quad \text{\{P\}C\{Q\}}}{\text{\{P'\}C\{Q\}}}\,?? \qquad \frac{\{P\}C\{Q\} \quad Q \vdash Q'}{\{P\}C\{Q'\}} \qquad \frac{\{I \wedge B\}C\{I\}}{\{I\}\texttt{while}\ B\ \texttt{do}\ \{I \wedge \neg B\}}$$

4. A terminating run of the interpreter will give us a **proof** of assertions at all program points.

# Implications for Overapproximating Analysis

$$\frac{P \vdash P' \quad \{P\}C\{Q\}}{\{P'\}C\{Q\}}\ ??$$

This rule is unsound

So, we generate "candidate" preconditions, then filter unsafe ones our using sound forwards analysis

`delete-doublestar` uses the usual C trick of double indirection to avoid unnecessary checking for the first element, when deleting an element from a list.

```
void delete-doublestar(nodeT **listP, elementT value)
{
  nodeT *currP, *prevP;
  prevP=NULL;
  for (currP=*listP; currP!=NULL; prevP=currP, currP=currP->next) {
    if (currP->element==value) {  /* Found it. */
      if (prevP==NULL) *listP=currP->next;
      else prevP->next=currP->next;
      free(currP);
} } }
```

`delete-doublestar` uses the usual C trick of double indirection to avoid unnecessary checking for the first element, when deleting an element from a list.

```
void delete-doublestar(nodeT **listP, elementT value)
{
  nodeT *currP, *prevP;
  prevP=NULL;
  for (currP=*listP; currP!=NULL; prevP=currP, currP=currP->next) {
    if (currP->element==value) {  /* Found it. */
      if (prevP==NULL) *listP=currP->next;
      else prevP->next=currP->next;
      free(currP);
} } }
```

What is the Footprint?

`delete-doublestar` uses the usual C trick of double indirection to avoid unnecessary checking for the first element, when deleting an element from a list.

```
void delete-doublestar(nodeT **listP, elementT value)
{
  nodeT *currP, *prevP;
  prevP=NULL;
  for (currP=*listP; currP!=NULL; prevP=currP, currP=currP->next) {
    if (currP->element==value) {  /* Found it. */
      if (prevP==NULL) *listP=currP->next;
      else prevP->next=currP->next;
      free(currP);
} } }
```

## What is the Footprint?

$$\texttt{listP} \mapsto \texttt{x\_} * \texttt{ls(x\_,x1\_)} * \texttt{x1\_} \mapsto \texttt{element:value},$$

$$\texttt{listP} \mapsto \texttt{x\_} * \texttt{ls(x\_,NULL)}$$

| Program | Time (s) | Memory | # of Disjuncts | Unsafe Pre | Discovered Precondition |
|---|---|---|---|---|---|
| append.c | 0.03501 | 0.74Mb | 4 | 0 | ls(x,NULL) |
| append-dispose.c | 0.09966 | 0.74Mb | 17 | 0 | ls(x,NULL)*ls(y,NULL) |
| copy.c | 0.03076 | 0.74Mb | 4 | 0 | ls(c,NULL) |
| create.c | 0.01370 | 0.49Mb | 1 | 0 | emp |
| delete-doublestar.c | 0.04521 | 0.49Mb | 10 | 0 | listP↦x_*ls(x_,x1_)*x1_↦element:value,<br>listP↦x_*ls(x_,NULL) |
| delete-all.c | 0.01357 | 0.49Mb | 4 | 0 | ls(c,NULL) |
| delete-all-circular.c | 0.01564 | 0.49Mb | 3 | 0 | c↦c_*ls(c_,c) |
| delete-lseg.c | 0.63947 | 1.23Mb | 48 | 0 | z≠NULL∧ls(c,z)*ls(z,NULL),<br>z≠w∧ls(c,z)*ls(z,w)*w↦NULL,<br>z≠w∧w_≠NULL∧ls(c,z)*ls(z,w)*w↦w_,<br>z≠c∧c↦NULL,<br>z≠c∧z≠c_∧c↦c_*ls(c_,NULL),<br>c=NULL∧emp |
| find.c | 0.05659 | 0.74Mb | 12 | 0 | ls(c,b)*b↦NULL,<br>b_≠NULL∧ls(c,b)*b↦b_,<br>b≠c∧b≠c_∧c↦c_*lseg(c_,NULL),<br>b≠c∧c↦NULL,<br>c=NULL∧emp |
| insert.c | 0.17049 | 0.74Mb | 10 | 0 | e1≠NULL∧e2≠NULL∧c_≠d_∧c↦c_*ls(c_,d_)*d↦dta:e3,<br>e1≠NULL∧e2≠NULL∧c_≠NULL∧c↦c_*ls(c_,NULL),<br>e1≠NULL∧c↦NULL,<br>e1≠NULL∧e2=NULL∧c↦c_*c_↦-,<br>e1=NULL∧c↦-,<br>c=NULL∧emp |
| merge.c | 0.56092 | 1.47Mb | 30 | 30 | — |
| reverse.c | 0.01965 | 0.74Mb | 4 | 0 | ls(c,NULL) |
| traverse-circ.c | 0.01322 | 0.49Mb | 3 | 0 | c↦c_*ls(c_,c) |

| Program | Time (s) | Memory | # of Disjuncts | Unsafe Pre |
|---|---|---|---|---|
| t1394Diag-CancelIrp.c | 0.08928 | 1.23Mb | 11 | 2 |
| t1394Diag-CancelIrpFix.c | 0.20461 | 1.23Mb | 10 | 0 |
| t1394Diag_PnpRemoveDevice | T/O | — | — | — |
| t1394-BusResetRoutine.c | 0.14924 | 1.23Mb | 4 | 0 |
| t1394-GetAddressData.c | 0.08692 | 1.23Mb | 9 | 2 |
| t1394-GetAddressDataFix.c | 0.08906 | 1.23Mb | 3 | 0 |
| t1394-IsochDetachCompletionRoutine.c | 1.76640 | 2.70Mb | 39 | 0 |
| t1394-SetAddressData.c | 0.06614 | 1.23Mb | 9 | 1 |
| t1394-SetAddressDataFix.c | 0.12242 | 1.23Mb | 9 | 0 |

**Table 2.** Experimental results from firewire device driver routines.

Stop