STORING BOUNDS INFORMATION

- *Store by pointer*
  - *Soft bounds (Hash-table approach)*
  - *CCured Approach (Separate Variable for bounds)*
- *Store by region (Baggy Bounds)*

**CONSIDER THE LIST STRUCTURE BELOW, MODIFIED TO STORE BOUNDS INFORMATION**

```
1: struct list
2: {
3:     struct list *next;
4:     bounds b_next; //bound information for the next node
5:     int x;
6: }
```

**PROBLEMS:**

- It becomes difficult to link non-transformed source library with transformed library.
- Need to transform the whole system (many libraries) which effectively slows down the system.

**CCured Approach**

No bounds in the actual structure. Instead maintain Shadow lists that store only the bounds information for a corresponding list structure.

```
1: struct list_shadow
2: {
3:     bounds b_next; //bound information for the next node
4:     struct list_shadow *next; //pointer to the next shadow node
5: }
```

**CONS PROGRAM:**
1: struct list *cons(struct list *l, int x)
2: {
3: struct list * t = malloc(...);
4: t->x = x;
5: t->next = l;
6: return t;
7: 
8: };

WITH BOUNDS CHECKING INFORMATION INCLUDED:

1: struct list *cons(struct list *l, int val, bounds b_l, struct list_shadow* l_s)
2: {
3: struct list * t = malloc(sizeof(t));
4: struct list_shadow *t_s = malloc(sizeof(t_s));
5: bounds b_t = {t, t+1};
6: boundscheck(b_t,t,t+1);
7: t->x = val;
8: boundscheck(b_t,t,t+1);
9: t_s->next = l_s;
10: t_s->b_next = b_l;
11: t->next = l;
12: return {t, t_s};
13: };

- l_s – Previous head of shadow needs to be passed
- t->next = l => Bounds check t(only t is dereferenced) and update bounds of t->next.

REPRESENTATION:

[Diagram of linked list and shadow list with bounds checking]
On walking l, l_shadow is also walked simultaneously.
• Invariant condition: On each pointer dereference, it is within bounds. So shadow list doesn’t get overwritten.

ASSIGNING Q TO P:

```
<table>
<thead>
<tr>
<th>b_p</th>
<th>b_q</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>q</td>
</tr>
<tr>
<td>p_s</td>
<td>q_s</td>
</tr>
</tbody>
</table>
```

```
p=q  b_p = b_q
     p=q
     p_s = q_s
```

• Now okay if the library walks over the linked list
• Insertion into linked list, is not okay yet
• Hence useful for Read-only libraries only

STORING BOUNDS WITH REGIONS

Jones and Kelley: Pointer Bounds are associated with region that the pointer points to. Referent Object Strategy

• h_tablelookup(p) in earlier case
• use bounds_lookup(p) instead

```
1:  int *p = malloc(n);
2:  bounds_store(p, p+n);
3:  bounds t = bounds_lookup(p);
4:  bcheck(t, p, p+1)
5:  *p = 0;
```

<table>
<thead>
<tr>
<th>Hash- table</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
</tr>
<tr>
<td>p</td>
</tr>
<tr>
<td>p+1</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

• But p++ finds nothing in the hash table
• xxx – Dead area
• 1111 – Allocated regions

\[
\begin{array}{c|c}
|1111xxx| & |1111ttttttx| \\
\hline
0 & 4GB \\
\end{array}
\]

• **Soln.:** Use special data structure called “Interval Trees” with O(log n) time for lookups & updates
• Need to check pointer arithmetic to make sure it doesn’t go out of bounds

```c
bounds t = bounds_lookup(p);
p++;
b_check(t,p,p)
p=buf;
for(…)
```

• \[p = p+(q-r);\] \[=>\] \[p = p+q\]
  - Compiler \[p = p-r\]
  - Re-arrangement \[\text{(Out of bounds)}\]

• Downside: O(log n) time for every pointer dereference (slow)