List Segment Auto-Discovery in a Symbolic Heap

Kamil Dudka

December 1, 2010

Agenda

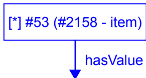
1. Symbolic Heap
2. List Segments – Abstraction
3. List Segments – Discovery

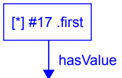
Symbolic Heap – Objects

[*] #53 (#2158 - item)

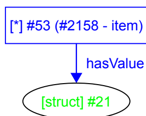
[*] #17 .first

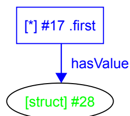
Symbolic Heap – **hasValue** Edges



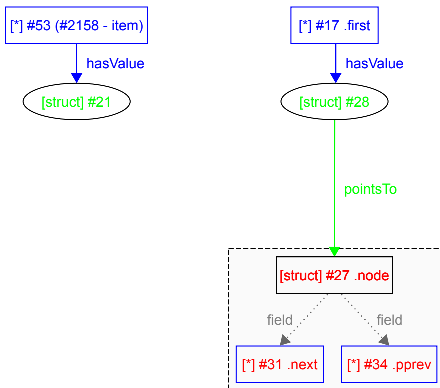


Symbolic Heap – Values

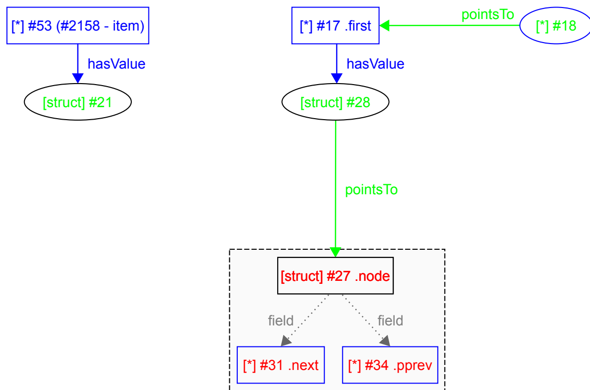




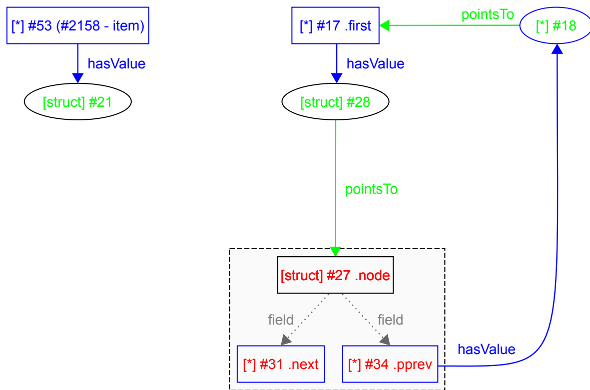
Symbolic Heap – pointsTo Edges



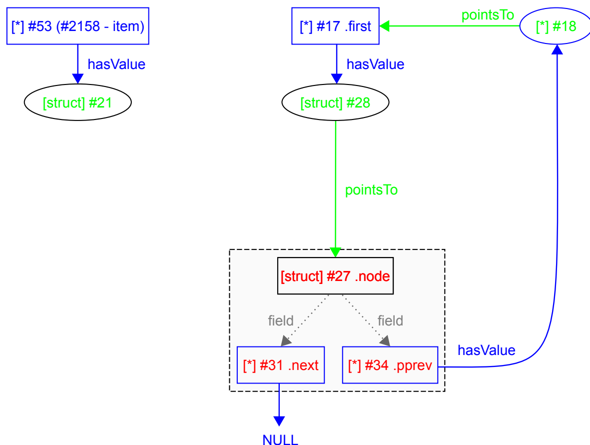
Symbolic Heap – **pointsTo** Edges



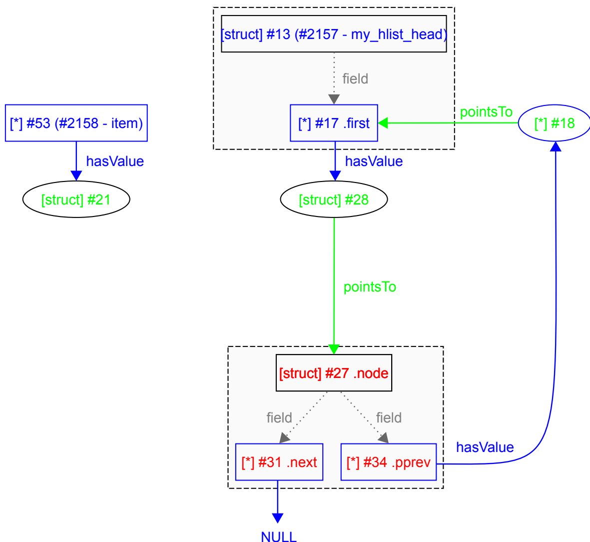
Symbolic Heap – **hasValue** Edges



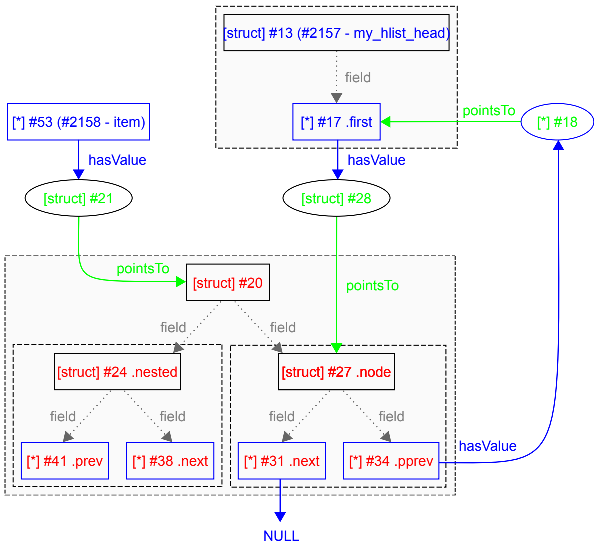
Symbolic Heap – **NULL** Value



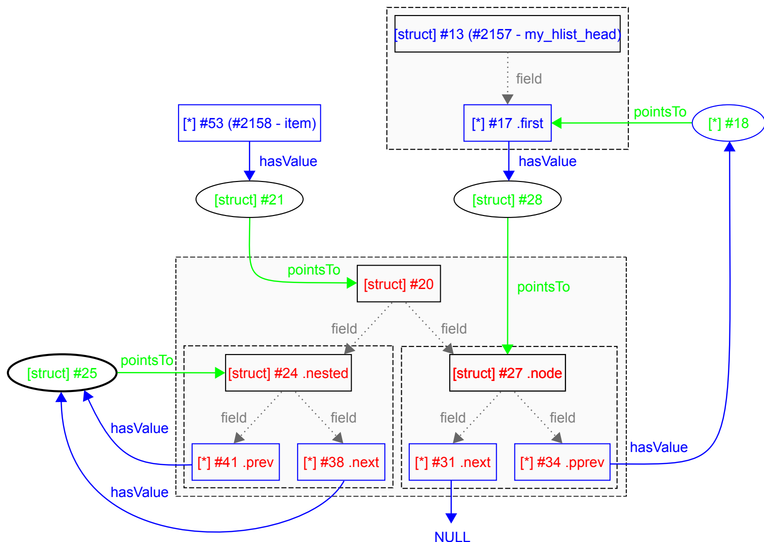
Symbolic Heap – Objects Composition



Symbolic Heap – Objects Composition



Symbolic Heap – Pointers to Self



Symbolic Heap – Example Was Generated by Code

```
1   #include <stdlib.h>
2   #include <linux/list.h>
3
4   struct my_hlist {
5       struct list_head nested;
6       struct hlist_node node;
7   };
8
9   int main() {
10      HLIST_HEAD(my_hlist_head);
11
12      struct my_hlist *item = malloc(sizeof *item);
13      INIT_LIST_HEAD(&item->nested);
14      INIT_HLIST_NODE(&item->node);
15      hlist_add_head(&item->node, &my_hlist_head);
16
17      ___sl_plot(NULL);
18      free(item);
19      return 0;
20  }
```

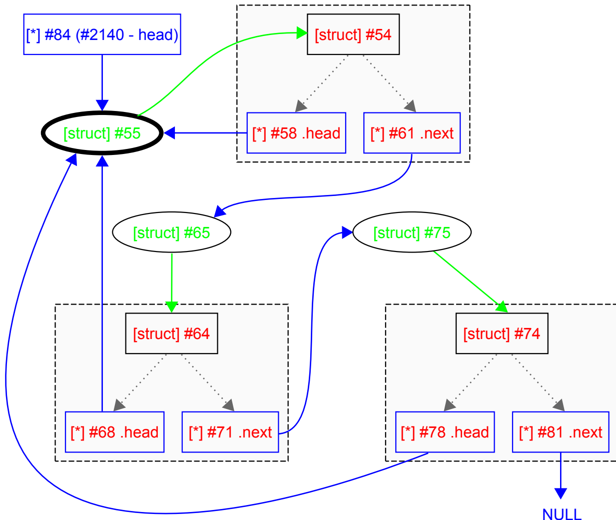
Agenda

1. Symbolic Heap
2. List Segments – Abstraction
3. List Segments – Discovery

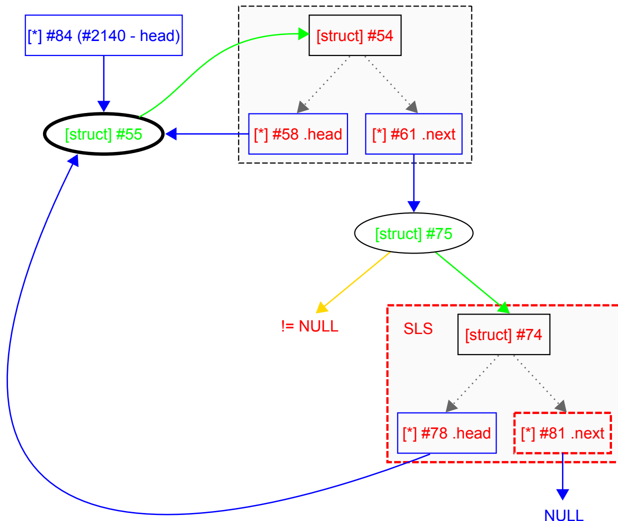
Example – SLL with Head – Creation

```
1   #include <stdlib.h>
2   #define NEW(type) (type *) malloc(sizeof(type))
3
4   struct item {
5       struct item *head;
6       struct item *next;
7   };
8
9   // create SLL of length 3 with head pointers
10  static struct item* create_sll(void)
11  {
12      struct item *head          = NEW(struct item);
13      head->head                 = head;
14      head->next                 = NEW(struct item);
15      head->next->head           = head;
16      head->next->next           = NEW(struct item);
17      head->next->next->head     = head;
18      head->next->next->next     = NULL;
19      return head;
20  }
```

Example – SLL with Head – Creation (1/2)



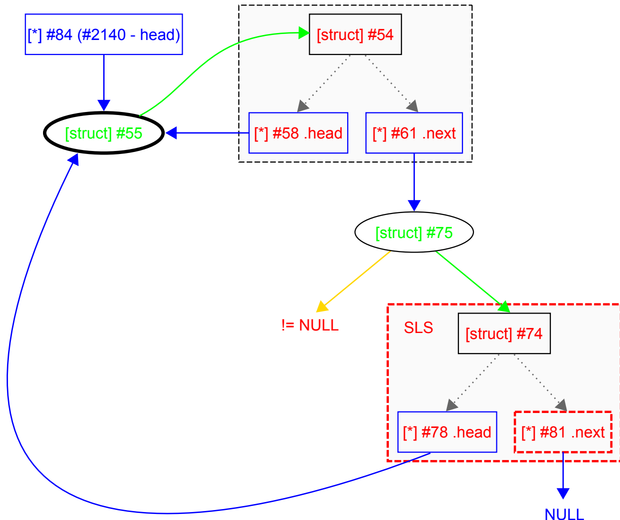
Example – SLL with Head – Creation (2/2)



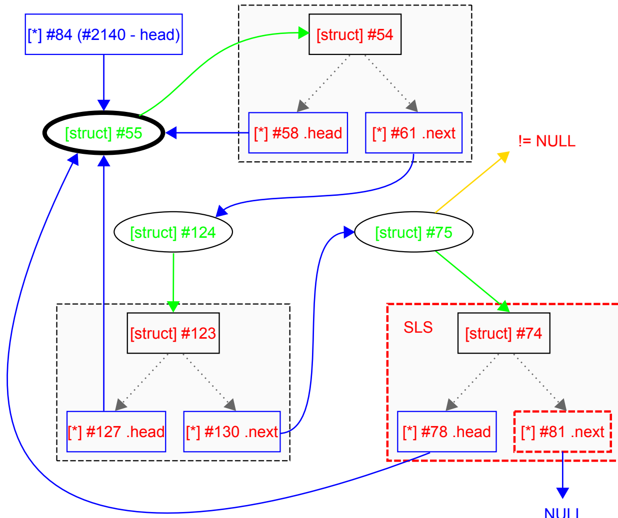
Example – SLL with Head – Insertion

```
1   // insert one node right after head and return its address
2   static struct item* create_longer_sll(void)
3   {
4       struct item *head        = create_sll();
5
6       struct item *next        = head->next;
7       head->next               = NEW(struct item);
8       head->next->head         = head;
9       head->next->next         = next;
10
11      return head->next;
12  }
```

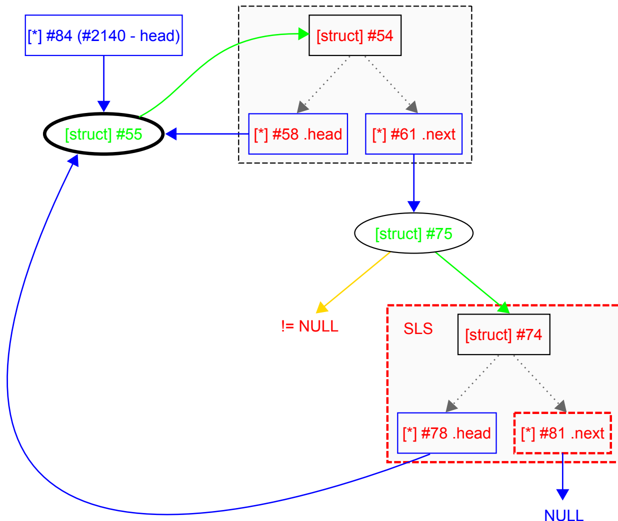
Example – SLL with Head – Insertion (1/4)



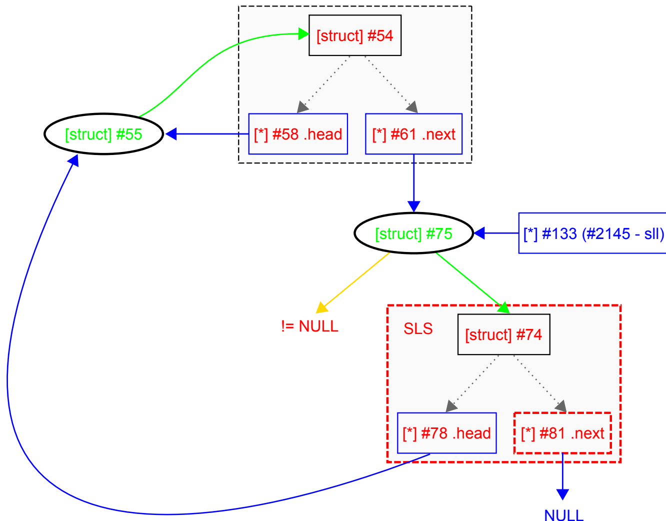
Example – SLL with Head – Insertion (2/4)



Example – SLL with Head – Insertion (3/4)



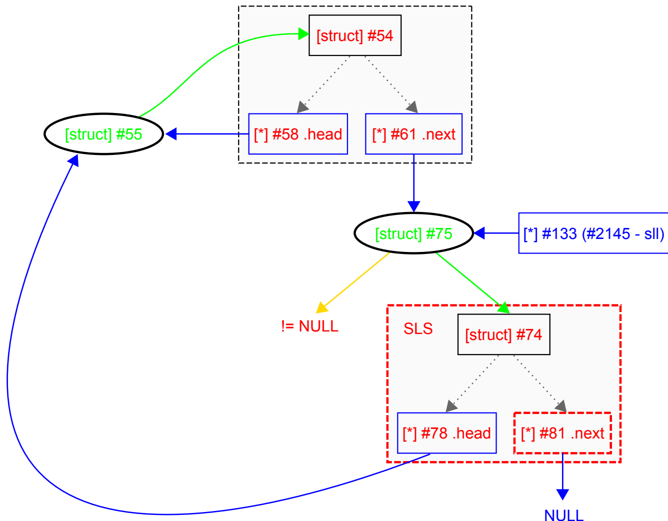
Example – SLL with Head – Insertion (4/4)



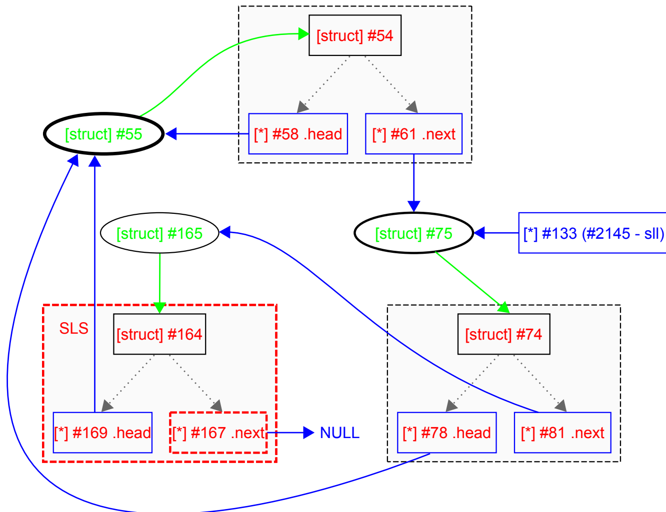
Example – SLL with Head – Destruction of Head

```
1  // destroy head
2  int main(void)
3  {
4      struct item *sll = create_longer_sll();
5      free(sll->head);
6      ___sl_plot_by_ptr(&sll, NULL);
7      return 0;
8  }
```

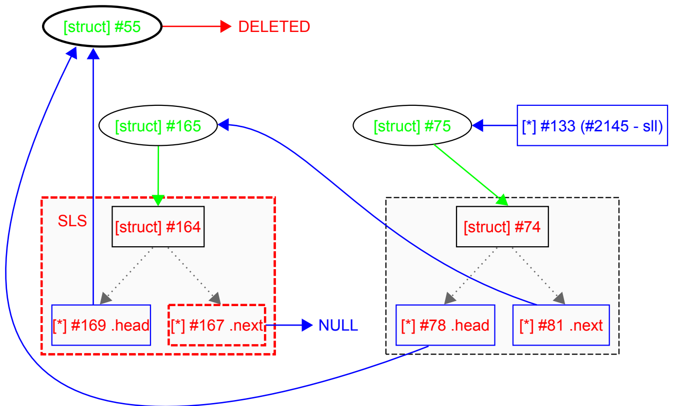
Example – SLL with Head – Destruction of Head (1/3)



Example – SLL with Head – Destruction of Head (2/3)

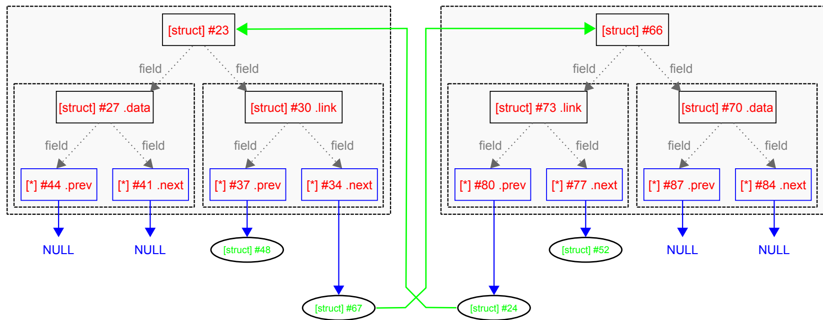


Example – SLL with Head – Destruction of Head (3/3)



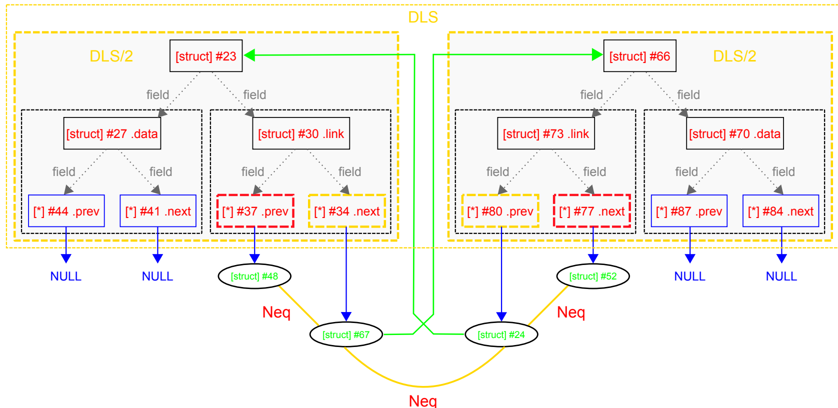
DLS – Doubly-linked List Segment (1/4)

DLS candidate



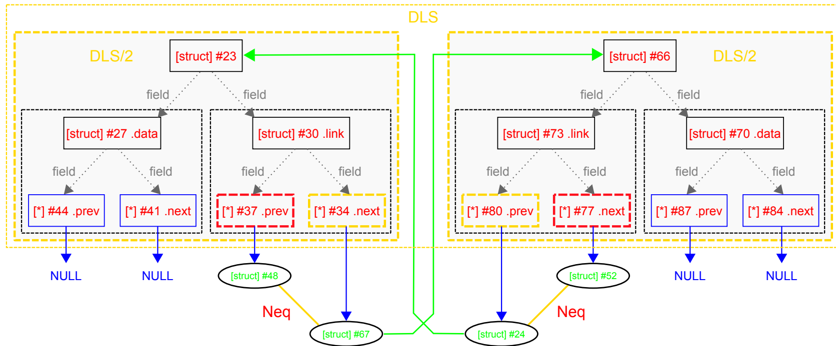
DLS – Doubly-linked List Segment (2/4)

DLS 2+



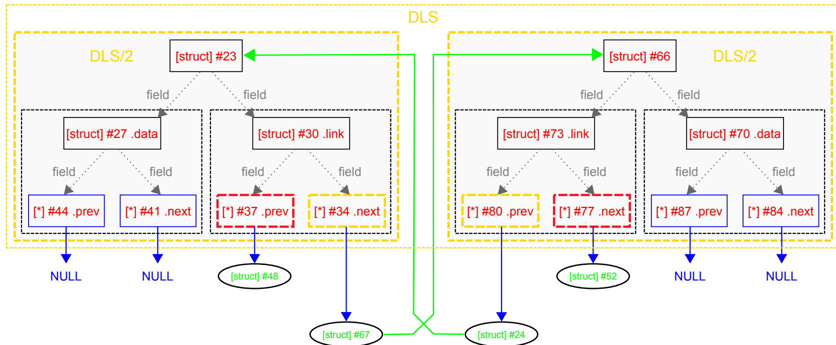
DLS – Doubly-linked List Segment (3/4)

DLS 1+



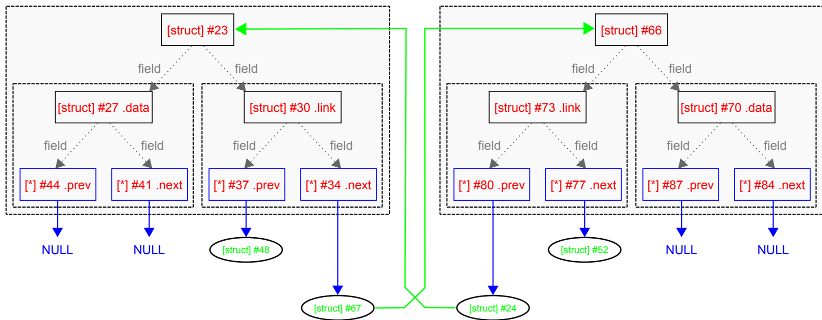
DLS – Doubly-linked List Segment (4/4)

DLS 0+



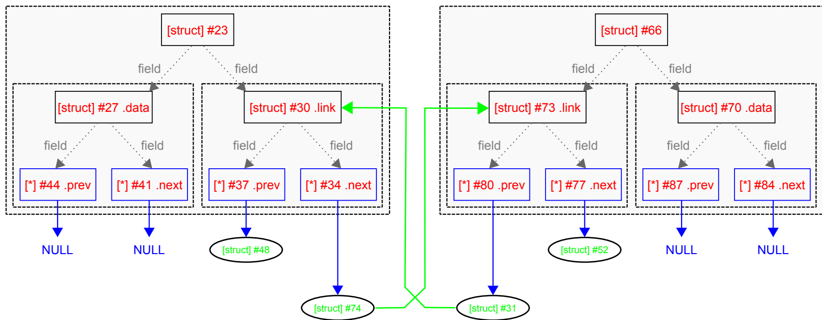
Linux DLS (1/4)

DLS candidate

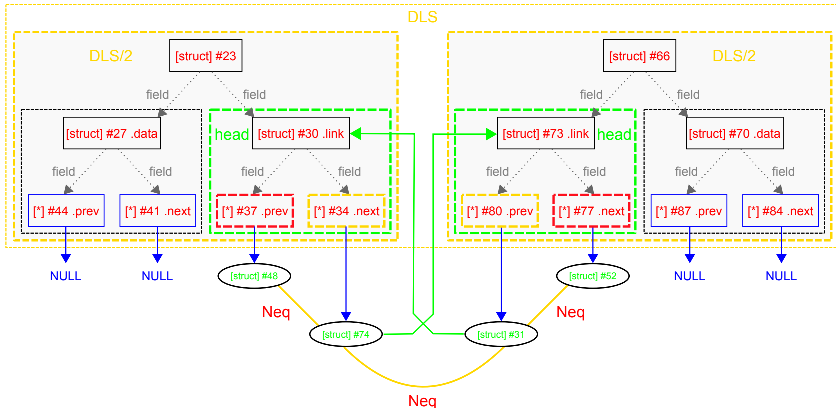


Linux DLS (2/4)

Linux DLS candidate

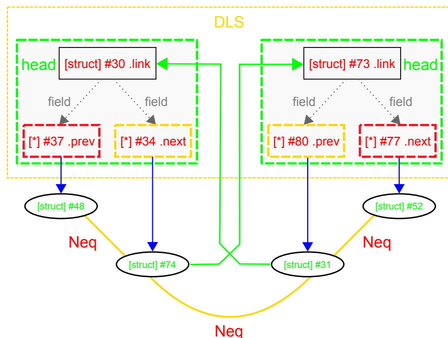


Linux DLS (3/4)



Linux DLS (4/4)

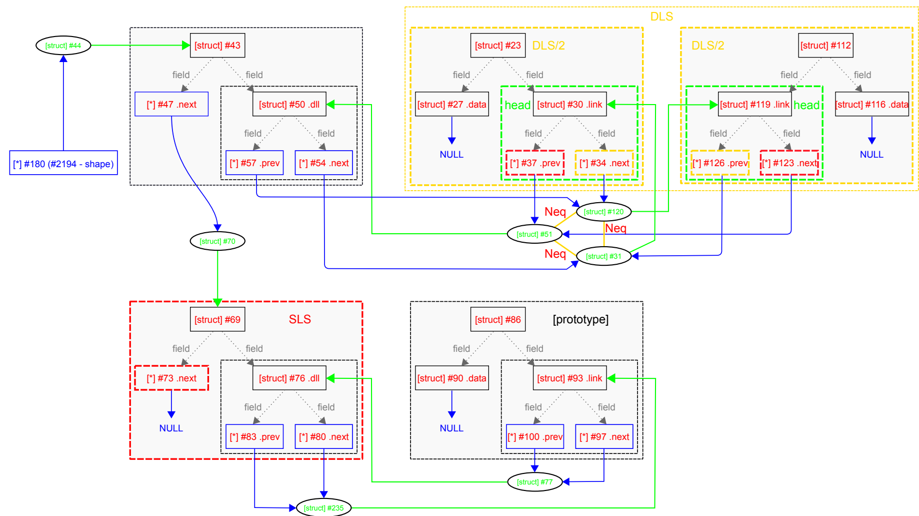
Linux DLS 2+ (heads only)



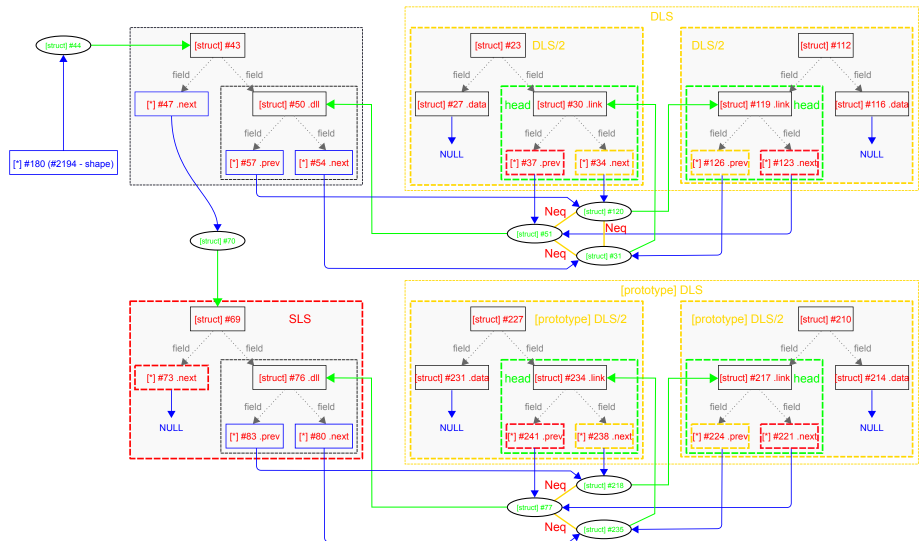
Agenda

1. Symbolic Heap
2. List Segments – Abstraction
3. List Segments – Discovery

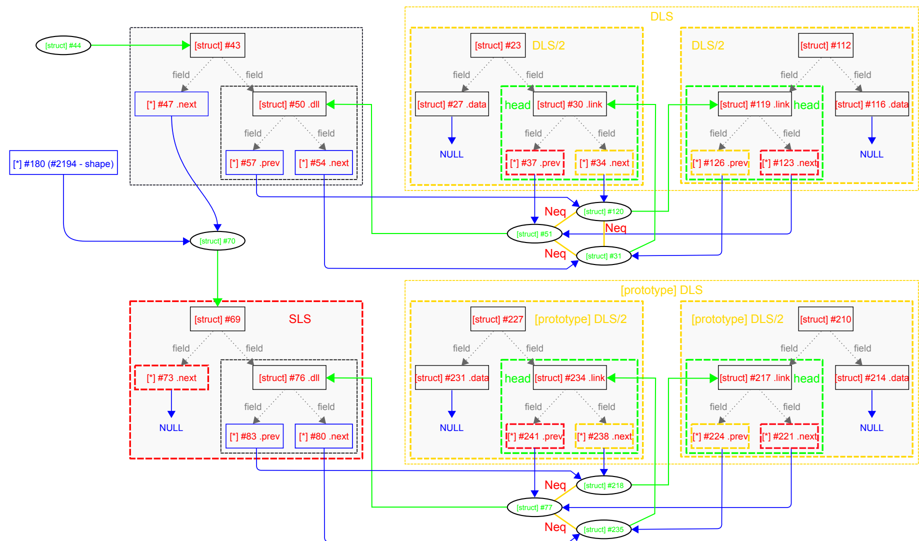
Example – SLS with nested Linux DLS(1/5)



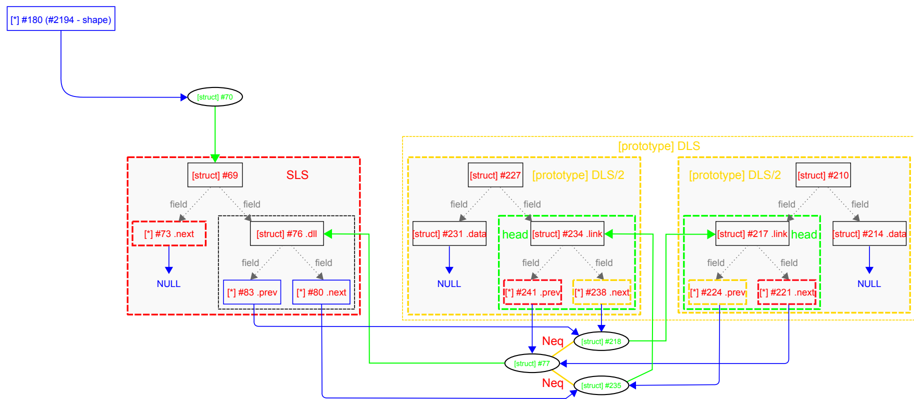
Example – SLS with nested Linux DLS(2/5)



Example – SLS with nested Linux DLS(3/5)



Example – SLS with nested Linux DLS(4/5)



Example – SLS with nested Linux DLS(5/5)

[*] #180 (#2194 - shape)

!= NULL

[struct] #70

SLS

[struct] #69

field

field

[*] #73 .next

NULL

[struct] #76 .dll

field

field

[*] #83 .prev

[*] #80 .next

[prototype] DLS

[prototype] DLS/2

[struct] #227

field

field

[struct] #231 .data

NULL

head

[struct] #234 .link

field

field

[*] #241 .prev

[*] #238 .next

[prototype] DLS/2

[struct] #210

field

field

[struct] #217 .link

head

[struct] #214 .data

NULL

field

field

[*] #224 .prev

[*] #221 .next

Neq

[struct] #218

[struct] #77

Neq

[struct] #235

List Segment Discovery – Top-Level Algorithm

1. Collect all entry candidates, each as a tuple:
   - `entry` – root object that may be a starting point of a list
   - `next` selector
   - `prev` selector (only for DLS)
   - `head` selector (only for Linux lists)

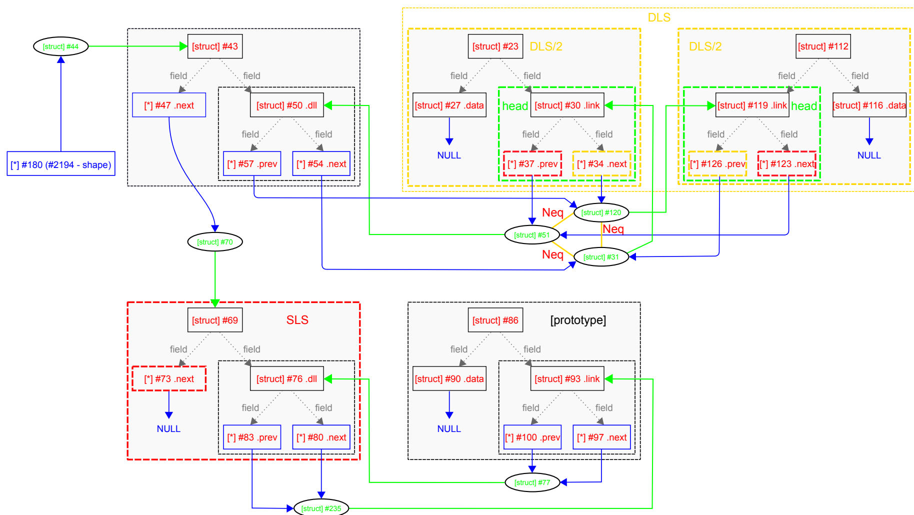
List Segment Discovery – Top-Level Algorithm

1. Collect all entry candidates, each as a tuple:
   - `entry` – root object that may be a starting point of a list
   - `next` selector
   - `prev` selector (only for DLS)
   - `head` selector (only for Linux lists)
2. Check maximum count of feasible abstraction steps – abstraction length – for each entry candidate.

List Segment Discovery – Top-Level Algorithm

1. Collect all entry candidates, each as a tuple:
   - `entry` – root object that may be a starting point of a list
   - `next` selector
   - `prev` selector (only for DLS)
   - `head` selector (only for Linux lists)
2. Check maximum count of feasible abstraction steps – abstraction length – for each entry candidate.
3. Return the longest possible abstraction as entry candidate coupled with the corresponding abstraction length.

Example – SLS with nested Linux DLS



List Segment Discovery – Core Algorithm (1/2)

1. Using `entry`, `next` and `head`, jump to the next object.

List Segment Discovery – Core Algorithm (1/2)

1. Using `entry`, `next` and `head`, jump to the next object.
2. Check validity of the next object, including:
   - availability of the next object
   - type of the next object
   - back-link in case of DLS
   - segment compatibility in case of already abstract object(s)

List Segment Discovery – Core Algorithm (1/2)

1. Using `entry`, `next` and `head`, jump to the next object.
2. Check validity of the next object, including:
   - availability of the next object
   - type of the next object
   - back-link in case of DLS
   - segment compatibility in case of already abstract object(s)
3. Try to join data (prototypes and shared data) in read-only mode.

List Segment Discovery – Core Algorithm (1/2)

1. Using `entry`, `next` and `head`, jump to the next object.
2. Check validity of the next object, including:
   - availability of the next object
   - type of the next object
   - back-link in case of DLS
   - segment compatibility in case of already abstract object(s)
3. Try to join data (prototypes and shared data) in read-only mode.
4. Validate pointing objects.

List Segment Discovery – Core Algorithm (1/2)

1. Using `entry`, `next` and `head`, jump to the next object.
2. Check validity of the next object, including:
   - availability of the next object
   - type of the next object
   - back-link in case of DLS
   - segment compatibility in case of already abstract object(s)
3. Try to join data (prototypes and shared data) in read-only mode.
4. Validate pointing objects.
5. Validate prototypes.

List Segment Discovery – Core Algorithm (1/2)

1. Using `entry`, `next` and `head`, jump to the next object.
2. Check validity of the next object, including:
   - availability of the next object
   - type of the next object
   - back-link in case of DLS
   - segment compatibility in case of already abstract object(s)
3. Try to join data (prototypes and shared data) in read-only mode.
4. Validate pointing objects.
5. Validate prototypes.
6. Increment abstraction length by one and go back to 1), using the next object as `entry`.

List Segment Discovery – Core Algorithm (2/2)

Some notes:

- A set of already traversed objects needs to be maintained.

List Segment Discovery – Core Algorithm (2/2)

Some notes:

- A set of already traversed objects needs to be maintained.
- An already existing DLS pair on the path is treated as one object.

List Segment Discovery – Core Algorithm (2/2)

Some notes:

- A set of already traversed objects needs to be maintained.
- An already existing DLS pair on the path is treated as one object.
- Cycles consisting only of segments are not allowed.

Conclusion (1/2)

What works already:

- singly- and doubly-linked lists

Conclusion (1/2)

What works already:

- singly- and doubly-linked lists
- cyclic/acyclic lists

Conclusion (1/2)

What works already:

- singly- and doubly-linked lists
- cyclic/acyclic lists
- Linux lists

Conclusion (1/2)

What works already:

- singly- and doubly-linked lists
- cyclic/acyclic lists
- Linux lists
- shared data

Conclusion (1/2)

What works already:

- singly- and doubly-linked lists
- cyclic/acyclic lists
- Linux lists
- shared data
- prototypes, including nested lists

Conclusion (1/2)

What works already:

- singly- and doubly-linked lists
- cyclic/acyclic lists
- Linux lists
- shared data
- prototypes, including nested lists

- *some* combinations of the things above

Conclusion (2/2)

Short-term schedule:

- optional prototypes

Conclusion (2/2)

Short-term schedule:

- optional prototypes
- Linux lists based on `hlist_head` / `hlist_node`

Conclusion (2/2)

Short-term schedule:

- optional prototypes
- Linux lists based on `hlist_head` / `hlist_node`
- handling of non-pointer data (`int`, `bool`, `enum`, etc.)