Verification of Low-Level List Manipulation (work in progress)

Kamil Dudka[1,2] Petr Peringer[1] Tomáš Vojnar[1]

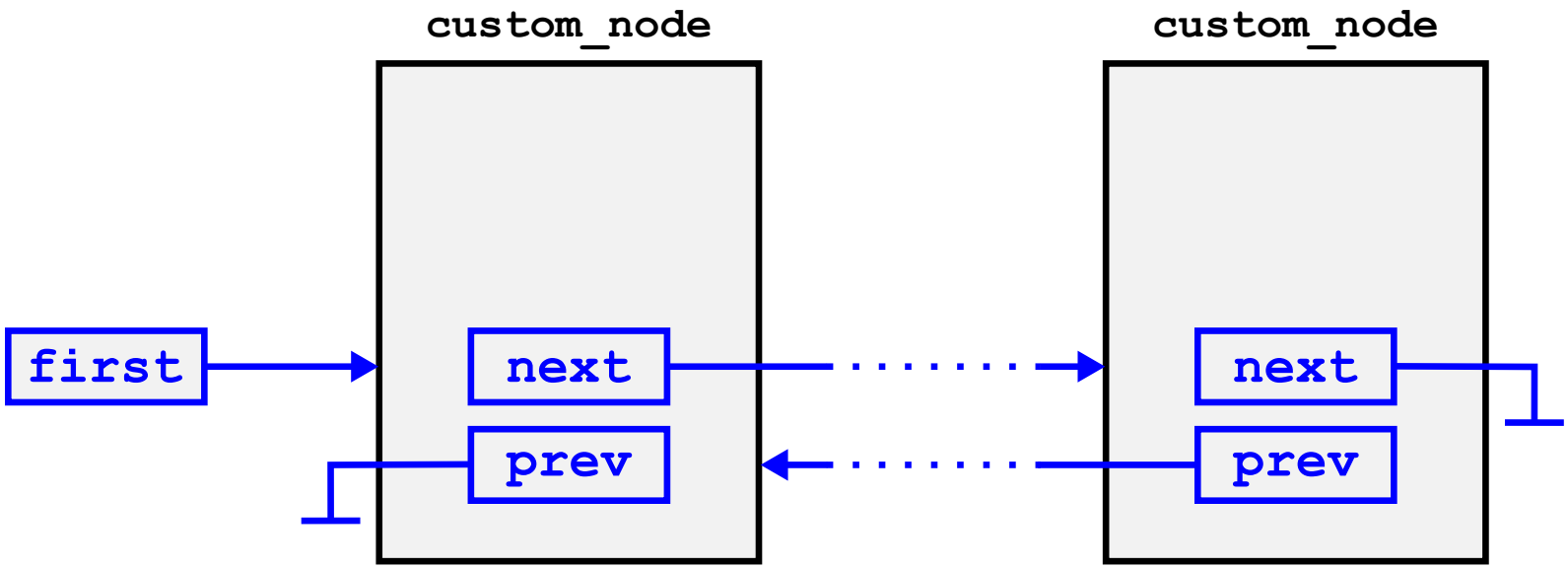
[1]FIT, Brno University of Technology, Czech Republic

[2]Red Hat Czech, Brno, Czech Republic

CP-meets-CAV, June 28, 2012

Low-level Memory Manipulation

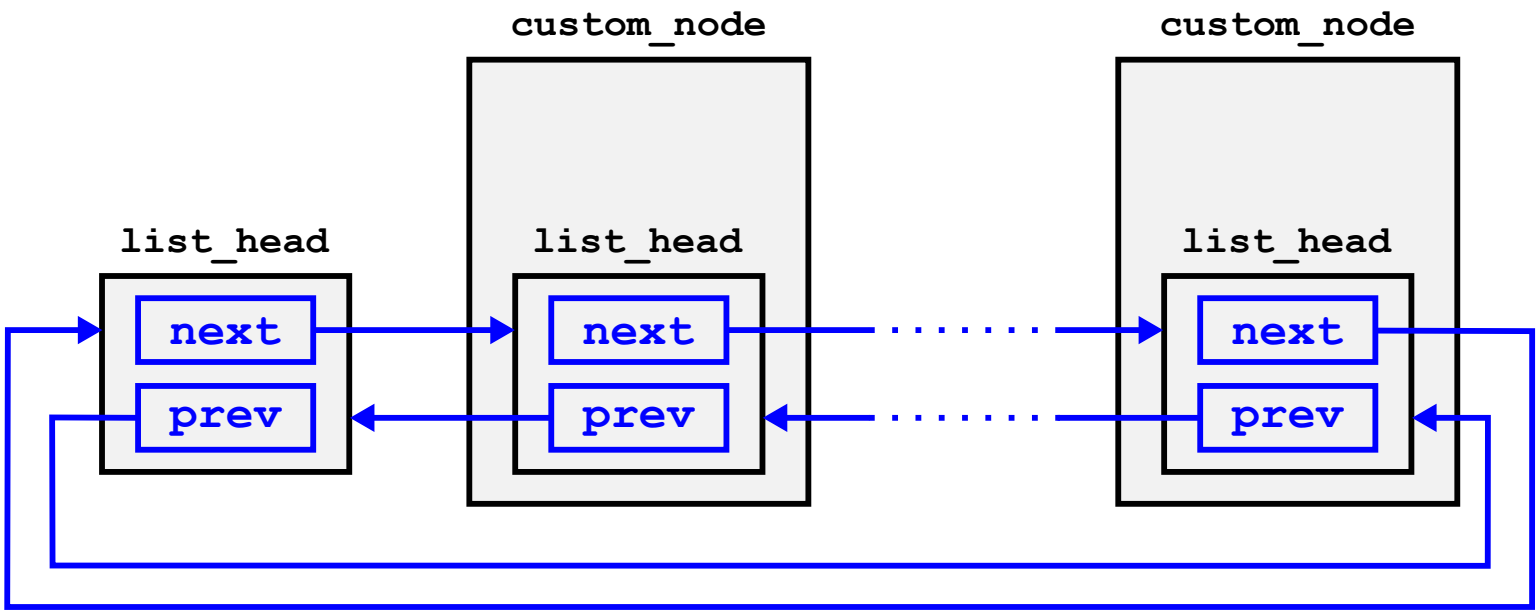
Doubly-Linked Lists: Textbook Style




```
struct custom_node {
    t_data data;
    struct custom_node *next;
    struct custom_node *prev;
};
```

Doubly-Linked Lists in Linux

- Cyclic, linked through pointers pointing inside list nodes.
- Pointer arithmetic used to get to the boundary of the nodes.
- Non-uniform: one node is missing the custom envelope.

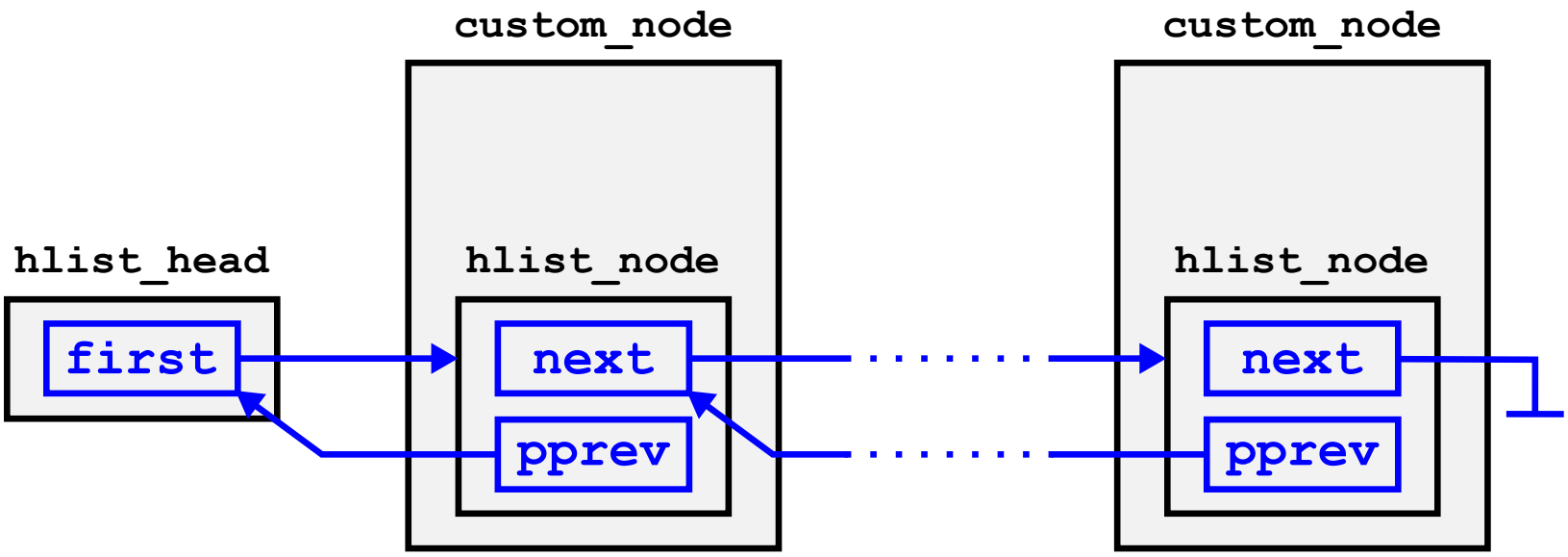



```
struct list_head {
    struct list_head *next;
    struct list_head *prev;
};
```

```
struct custom_node {
    t_data data;
    struct list_head head;
};
```

Linux Lists: Optimised for Hash Tables

- Using pointers to pointers to save 8 bytes (for 64b addressing) in the head nodes stored in hash tables.




```
struct hlist_node {
    struct hlist_node *next;
    struct hlist_node **pprev;
};
```

```
struct custom_node {
    t_data data;
    struct hlist_node node;
};
```

Linux Lists: Traversal

- ... as seen by the programmer:

```
list_for_each_entry(pos, list, head)
    printf(" %d", pos->value);
```

- ... as seen by the compiler:

```
for(pos = ((typeof(*pos) *)((char *)(list->next)
  -(unsigned long)(&((typeof(*pos) *)0)->head)));
  &pos->head != list;
  pos = ((typeof(*pos) *)((char *)(pos->head.next)
  -(unsigned long)(&((typeof(*pos) *)0)->head))))
{
    printf(" %d", pos->value);
}
```

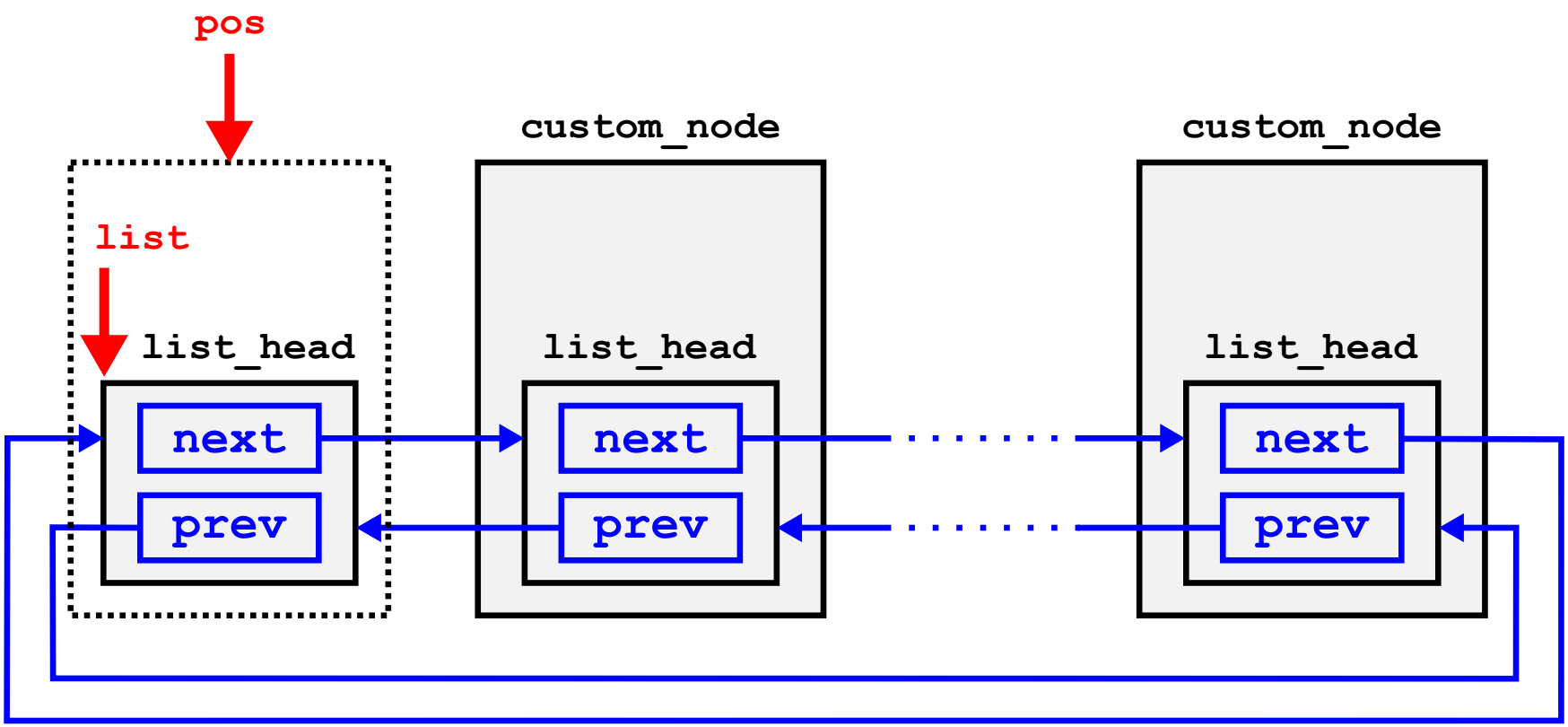
- ... as seen by the analyser (assuming 64b addressing):

```
for(pos = (char *)list->next - 8;
    &pos->head != list;
    pos = (char *)pos->head.next - 8)
{
    printf(" %d", pos->value);
}
```

Linux Lists: End of the Traversal

- Correct use of pointers pointing outside of allocated memory:

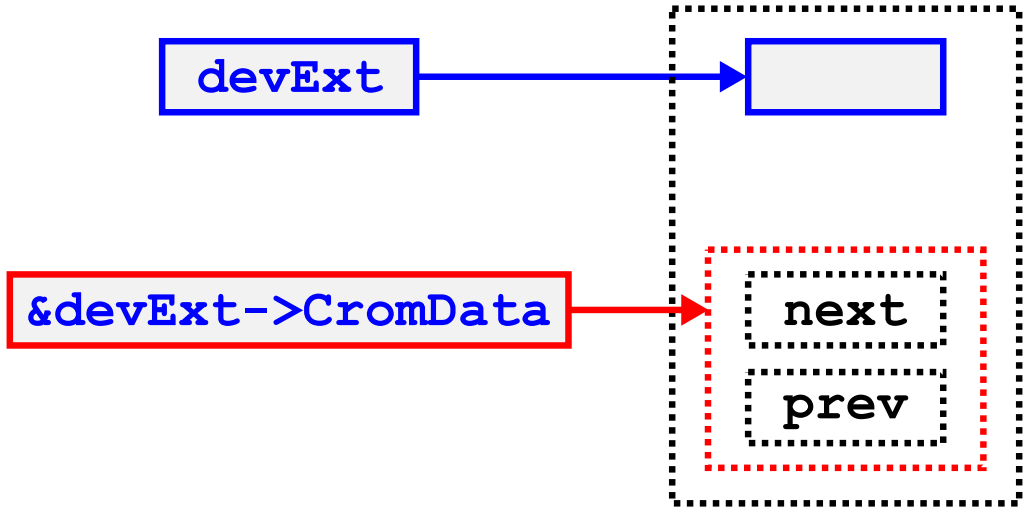
```
&pos->head != list;
```



Tracking the Block Size

- When not tracking block sizes, many errors may be missed:

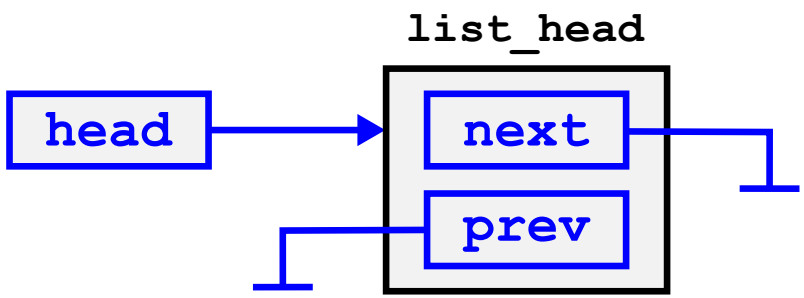
```
typedef struct _DEVICE_EXTENSION {
    PDEVICE_OBJECT      PortDeviceObject;
    // ...
    LIST_ENTRY          CromData;
    // ...
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;

PDEVICE_EXTENSION devExt = (PDEVICE_EXTENSION)
    malloc(sizeof(PDEVICE_EXTENSION));
InitializeListHead(&devExt->CromData);
```



Tracking Nullified Blocks

- Large chunks of memory are often nullified at once, their fields are gradually used, the rest must stay null.

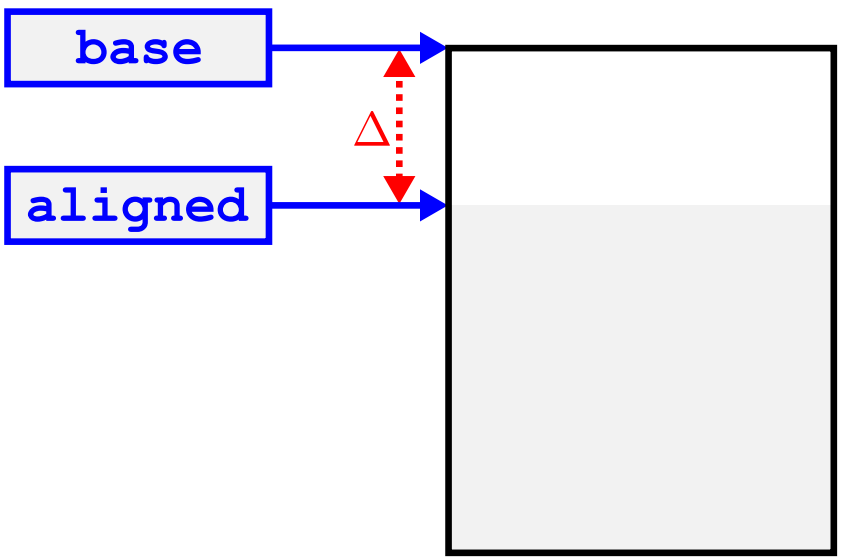
```
struct list_head {
    struct list_head *next;
    struct list_head *prev;
};

struct list_head *head = calloc(1U, sizeof *head);
```



Alignment of Pointers

- Alignment of pointers implies a need to deal with pointers whose target is given by an interval of addresses:

```
aligned = ((unsigned)base + mask) & ~mask;
```



$\texttt{mask} = 2^N - 1, \quad N \geq 0$

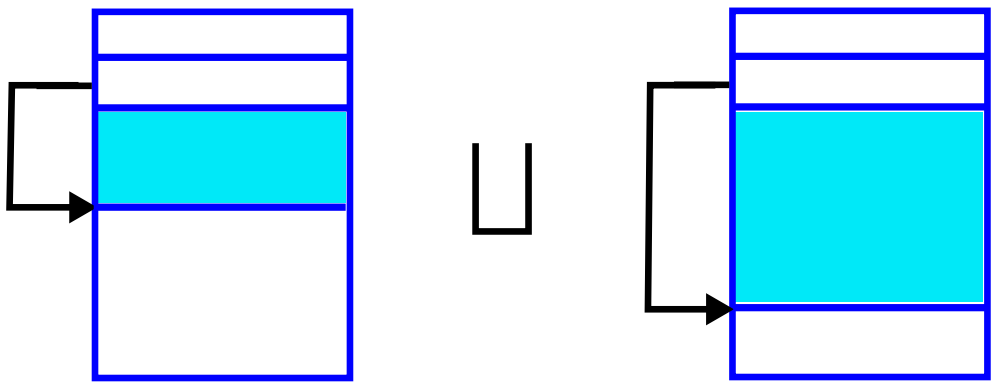
$0 \leq \Delta \leq \texttt{mask}$

e.g. alignment on multiples of 8

$$\begin{aligned} \texttt{mask} &= 0111 = 2^3 - 1 \\ \texttt{base} &= 0001 \\ \texttt{aligned} &= 1000 \end{aligned}$$

Pointers Arriving to Different Offsets

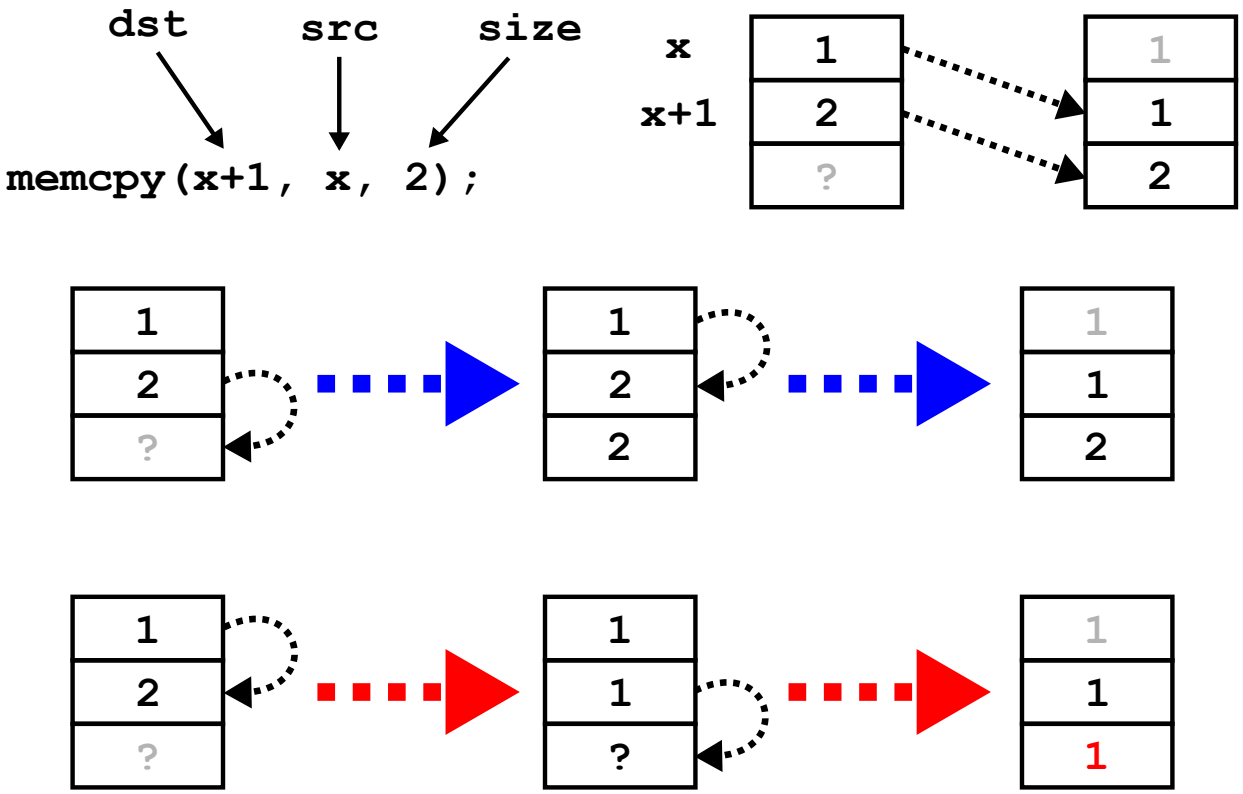
- Intervals of addresses arise also when joining blocks of memory with corresponding pointers arriving to different offsets.
  - Common, e.g., when dealing with sub-allocation.



- Moreover, when dealing with lists of blocks of different sizes, one needs to use blocks of interval size in order to be able to make the computation terminate.

Block Operations

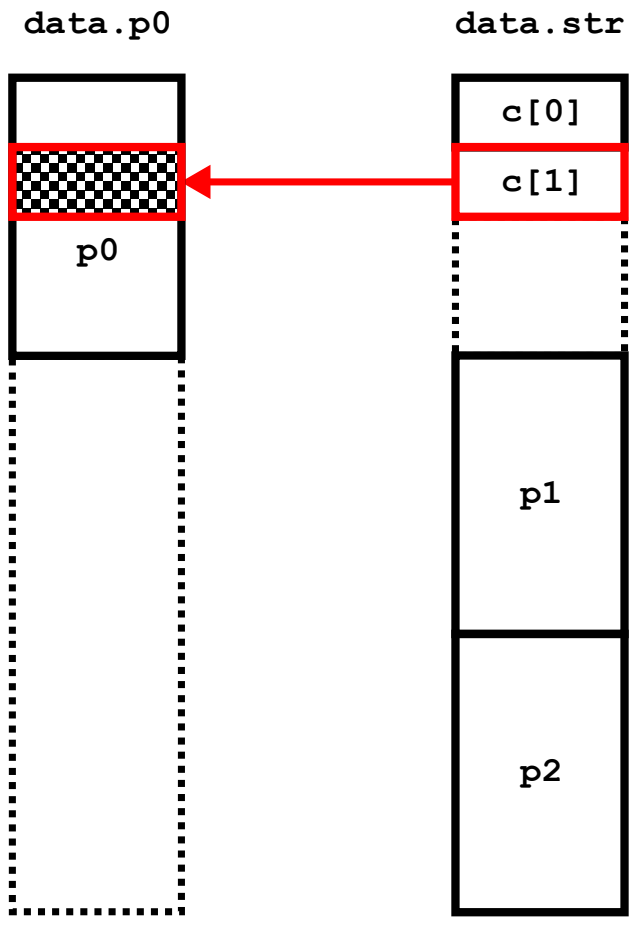
- Low-level code often uses block operations:
  `memcpy()`, `memmove()`, `memset()`, `strcpy()`.
- Incorrect use of such operations can lead to nasty errors – e.g., `memcpy()` and overlapping blocks:



Data Reinterpretation

- Due to unions, typecasting, or block operations, the same memory contents can be interpreted in different ways.

```
union {
    void *p0;
    struct {
        char c[2];
        void *p1;
        void *p2;
    } str;
} data;

// allocate 37B on heap
data.p0 = malloc(37U);
// introduce a memory leak
data.str.c[1] =
    sizeof data.str.p1;
// invalid free()
free(data.p0);
```



Predator

Predator: An Overview

- In principle based on separation logic with higher-order list predicates, but using a graph encoding of sets of heaps.
- Verification of low-level system code (in particular, Linux code) that manipulates dynamic data structures.
- Looking for memory safety errors (invalid dereferences, double free, buffer overrun, memory leaks, ...).
- Implemented as an open source gcc plugin:

```
http://www.fit.vutbr.cz/research/groups/verifit/tools/predator
```

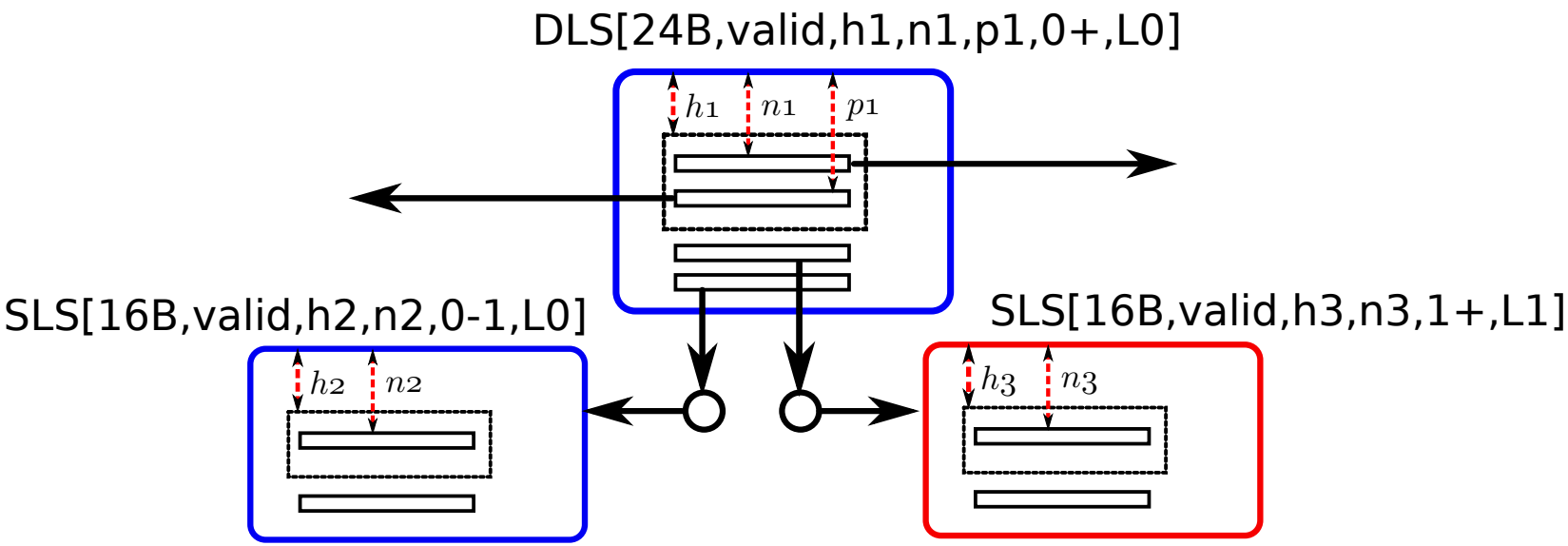
Symbolic Memory Graphs (SMGs)

In Predator, sets of memory configurations are represented using symbolic memory graphs (SMGs), together with a mapping from program variables to nodes of SMGs:

- SMGs are oriented graphs with two main types of nodes: objects (allocated space) and values (addresses, integers).
- Objects are further divided into:
  - regions, i.e., individual blocks of memory,
  - optional regions, i.e., either a region or null, and
  - singly-linked and doubly-linked list segments (SLSs/DLSs).
- Each object has some size in bytes and a validity flag.
  - Invalid (i.e., deallocated) objects are kept till somebody points to them to allow for pointer arithmetic and comparison over them.
- Explicit non-equality constraints on values are tracked.

Doubly-Linked List Segments

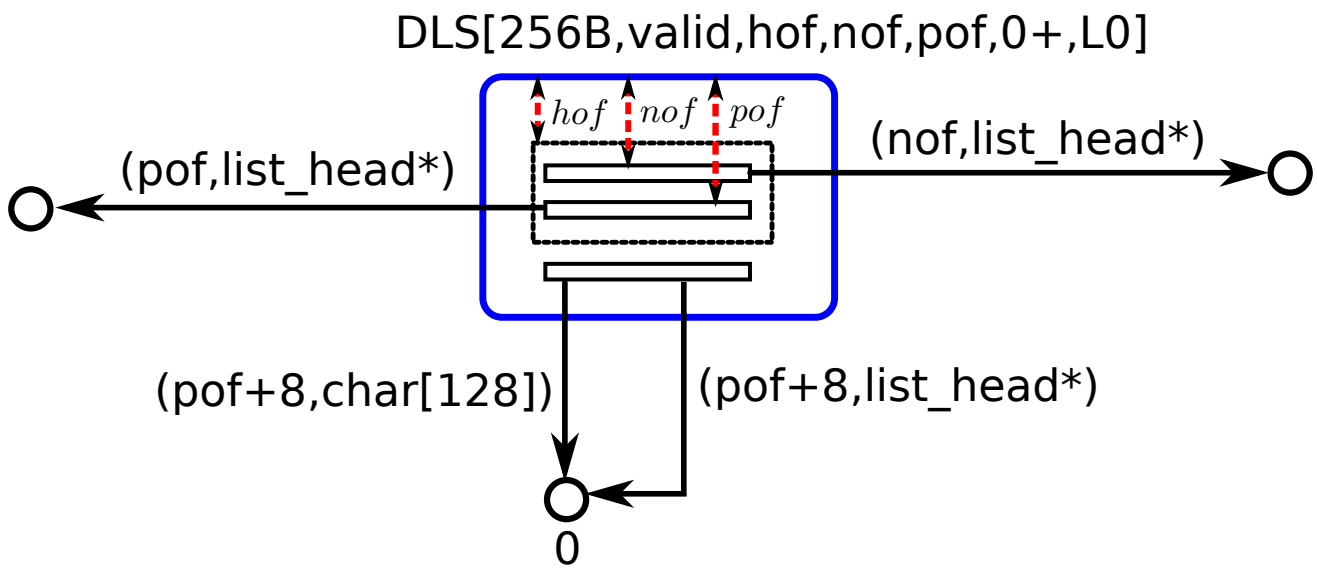
- Each DLS is given by a head, next, and prev field offset.
- DLSs can be of length $N+$ for any $N \geq 0$ or of length 0–1.
- Nodes of DLSs can point to objects that are:
  - shared: each node points to the same object,
  - nested: each node points to a separate copy of the object.
    - Implemented by tagging objects by their *nesting level*.



Has-Value Edges of SMGs

Has-value edges lead from objects to values and are labelled by:

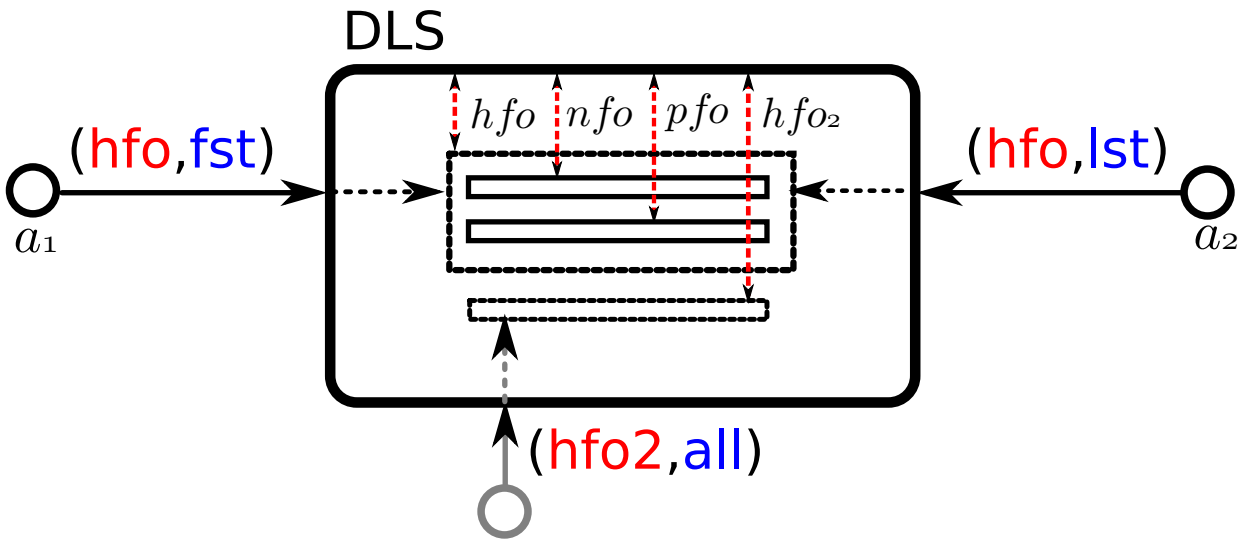
- the field offset, i.e., the offset of a value in an object, and
- the type of the value.
  - Due to reinterpretation, values of more types can be stored at the same offset.



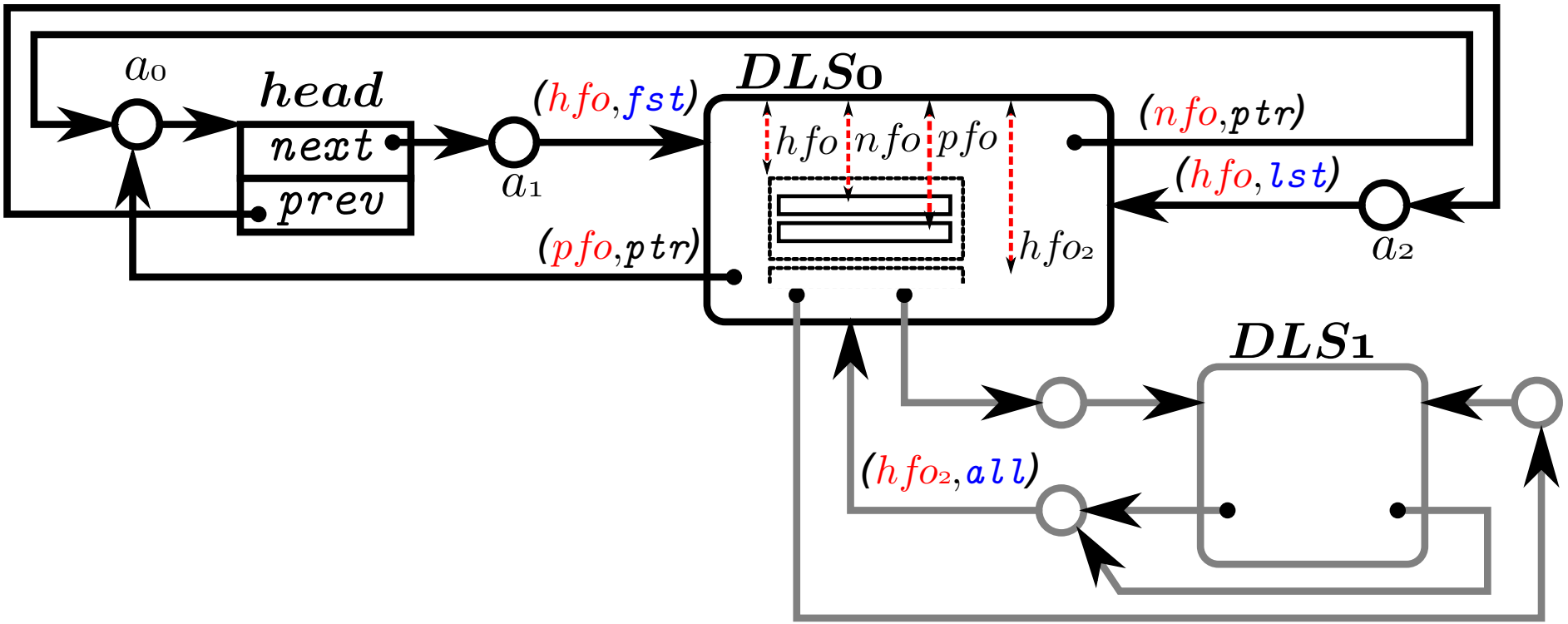
Points-to Edges of SMGs

Points-to edges lead from values (addresses) to objects and are labelled by

- the target offset and
- the target specifier which for a list segment says whether the pointer points to:
  - the *first node*,
  - the *last node*, or
  - *each node* (for edges going from nested objects).



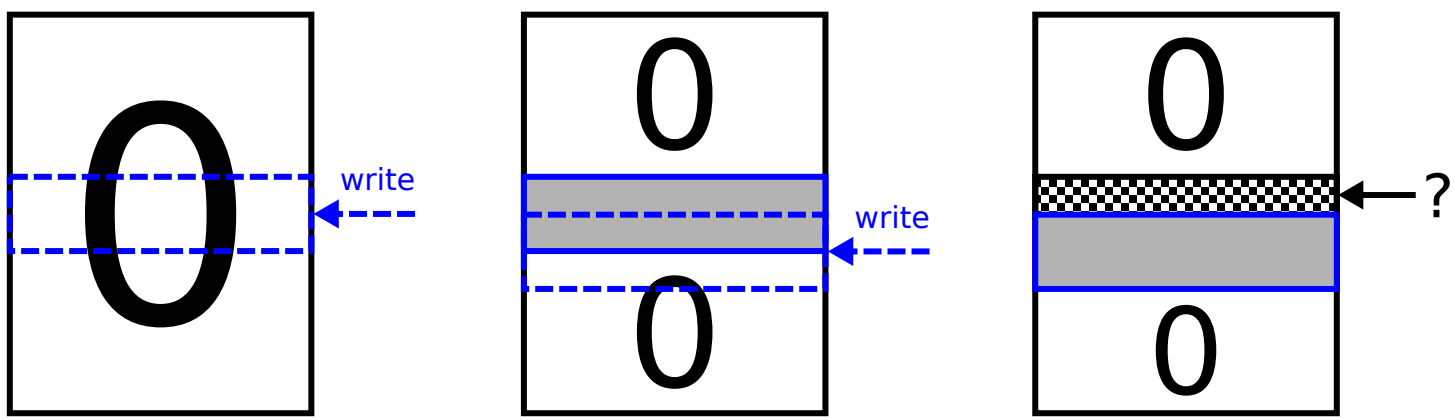
An SMG for Linux cDLLs of cDLLs



Data Reinterpretation

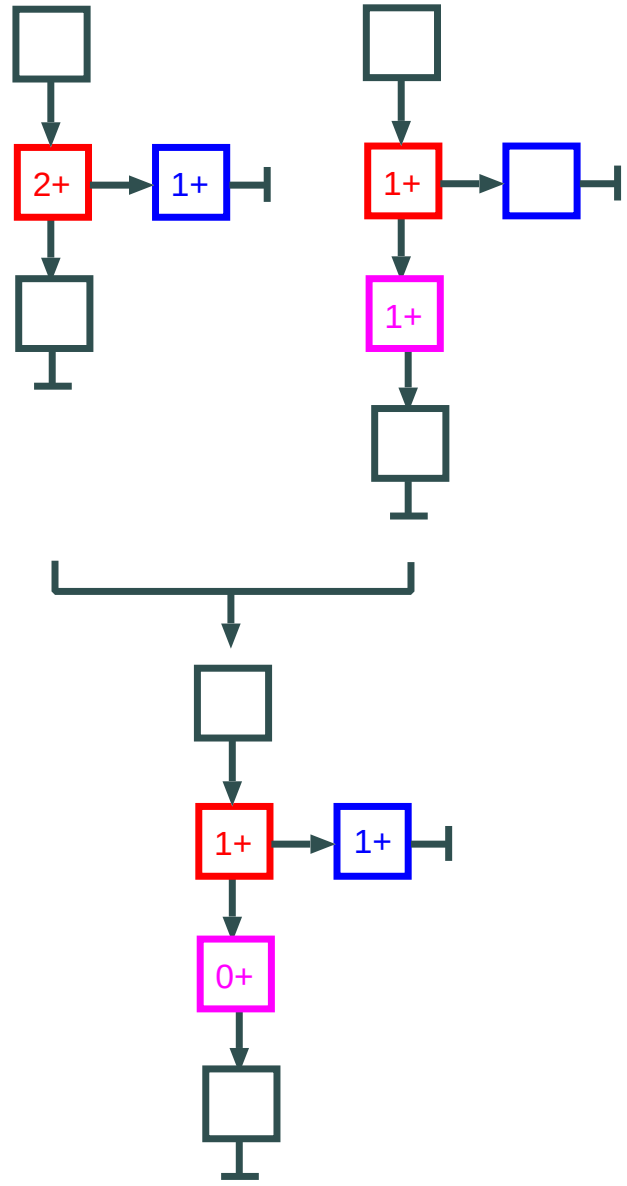
- Upon reading: a field with a given offset and type either exists, or an attempt to synthesise if from other fields is done.
- Upon writing: a field with a given offset and type is written, overlapping fields are adjusted or removed.
- Currently, for nullified/undefined fields of different size only.

```
// Allocating a nullified block and writing to it.
char *buffer = calloc(1, 64);
void **ptr1 = buffer + 30; *ptr1 = buffer;
void **ptr2 = buffer + 32; *ptr2 = buffer;
```



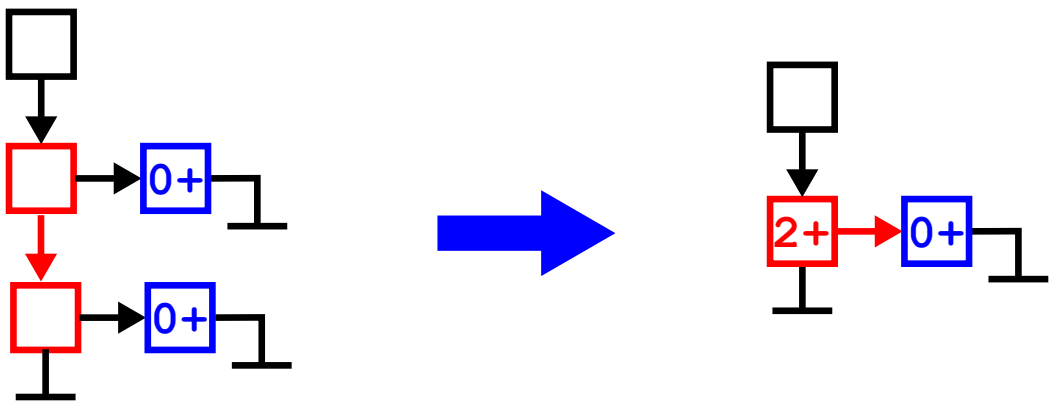
Join Operator: The Main Idea

- Traverses two SMGs and tries to join simultaneously encountered objects.
- Regions with the same *size*, *level*, *validity*, and the same defined *address fields* are joint using *reinterpretation*.
- DLSs can be joint with regions or DLSs under the same conditions as above + they must have the same *head*, *next*, and *prev offsets* (likewise for SLSs).
  - The length constraint has to be adjusted.
- If the above fails, try to insert an SLS/DLS of length 0+ or 0–1 into one of the heaps.
- Keep only shared non-equality constraints.



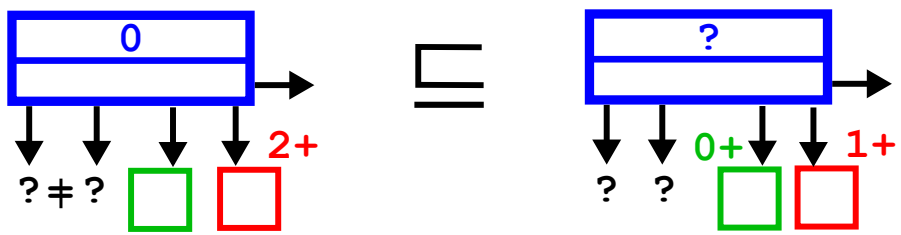
Abstraction: The Main Idea

- Based on collapsing uninterrupted sequences of objects into SLSs or DLSs.
- Starts by identifying sequences of valid objects that
  - have the same *size*, *level*, and defined *address fields* and
  - are *singly* / *doubly-linked* through fields at the same offset.
  - Can be refined by also considering *C-types of the objects* (if available).
- Uses join on the sub-heaps of such nodes to see whether their sub-heaps are compatible too.
  - Distinguishes cases of *shared and private sub-heaps*.



Controlling the Abstraction (1)

- There may be more sequences that can be collapsed.
  - We select among them according to their cost given by the *loss of precision* they generate.
- Three different costs of joining objects are distinguished:
  0. Joining *equal objects*:
     - Equal sub-heaps, same constraints on non-address and undefined address fields (via reinterpretation).
  1. One object semantically *covers* the other:
     - It has a more general sub-SMG, less constrained non-address and undefined address fields.



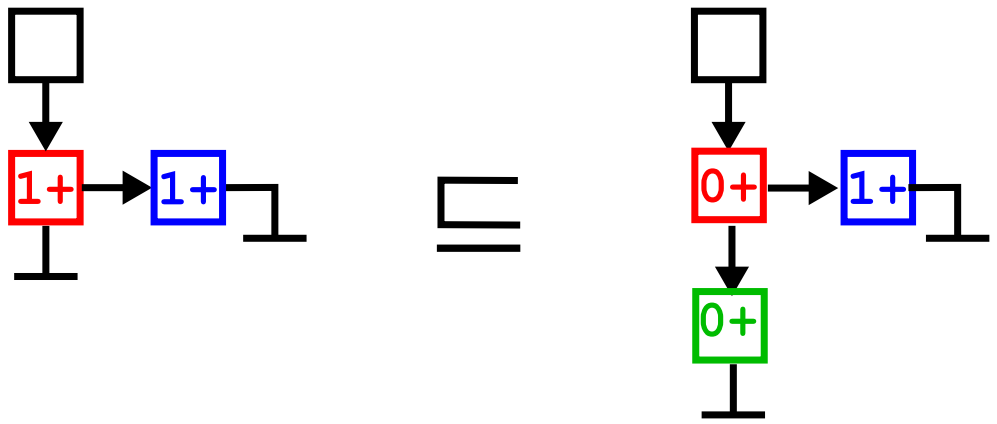
  2. None of the objects covers the other.

Controlling the Abstraction (2)

- For each object, find the maximal collapsing sequences (i.e., sequences which cannot be further extended).
- For the smallest cost for which one can collapse a sequence of at least some pre-defined minimum length, choose one of the longest sequences for that cost.
- Repeat till some sequence can be collapsed.

Entailment Checking

- The join of SMGs is again used:
  - It is checked that whenever non-equal objects are joint, less general objects always appear in the SMG to be entailed.



Predator: Case Studies (1)

- More than 256 case studies in total.
- Programs dealing with various kinds of lists (Linux lists, hierarchically nested lists, ...).
  - Concentrating on typical constructions of using lists.
  - Considering various typical bugs that appear in more complex lists (such as Linux lists).
- Correctness of pointer manipulation in various sorting algorithms (Insert-Sort, Bubble-Sort, Merge-Sort).
- We can also successfully handle the driver code snippets available with Slayer.
- Tried one of the drivers checked by Invader.
  - Found a bug caused by the test harness used, which is related to Invader not tracking the size of blocks.

Predator: Case Studies (2)

Verification of selected features of the following systems:

- The memory allocator from Netscape Portable Runtime (NSPR).
  - One size of the arenas for user allocation.
  - Allocation of blocks not exceeding the arena size for now.
- Logical Volume Manager (lvm2).
  - The (so far quite restricted) test harness uses doubly-linked lists instead of hash tables, which we do not support yet.

Predator: Future Work

- Further improve the support of interval-sized blocks and pointers with interval-defined targets.
  - Allow joining of blocks of different size.
  - Allow a richer set of program statements on interval-defined pointers.
  - Add more complex constraints on the intervals.
  - ...
- Support for additional shape predicates:
  - trees,
  - array segments,
  - ...
- Support for non-pointer data (mainly integers) stored in the data structures.

Related Tools

Many tools for verification of programs with dynamic linked data structures are currently under development. The closest to Predator are probably the following ones:

- Space Invader: pioneering tool based on separation logic (East London Massive: C. Calcagno, D. Distefano, P. O'Hearn, H. Yang).
- Slayer: a successor of Invador from Microsoft Research (J. Berdine, S. Ishtiaq, B. Cook).
- Forester: based on forest automata combining tree automata and separation (J. Šimáček, L. Holík, A. Rogalewicz, P. Habermehl, T. Vojnar).