

Automatizovaná formální verifikace v operačních systémech

Kamil Dudka

11. března 2010

- efektivní techniky pro verifikaci programů, které pracují s dynamickými datovými strukturami
- na vstupu bude reálný software z oblasti operačních systémů
- využití **separační logiky**

- přístupy založené na různých formalismech (automaty, logiky)
 - TVLA (Three-Valued Logic Analyzer)
 - PALE (Pointer Assertion Logic Engine)
 - separační logika
 - ...
- **separační logika** se vyznačuje největší škálovatelností z dosud navržených přístupů
- práce zaměřené na programy se seznamy seznamů:
 - [Reynolds, 2002]
 - [Berdine *et al.*, 2005]
 - [Distefano *et al.*, 2006]
- experimenty s nástrojem **Invader**

- vágní popis některých aspektů verifikace (např. rozšíření pro obousměrně vázané seznamy)
- implementace se výrazně liší od popisu
- vstupem verifikačních nástrojů obvykle nejsou reálné programy (a není jasné, jak byly z reálných programů získány)
- některé věci v implementaci nejsou pokryty v literatuře vůbec:
 - částečná práce s daty uloženými v seznamech
 - ukazatele dovnitř struktur
 - nástroj záhadně selhává
 - ...

Návaznost na existující práce

- s ohledem na výše uvedené jsme se rozhodli pokusit se výsledky zreprodukovat
- cílem je vytvořit si odrazový můstek pro další činnosti
- výsledkem je prozatím částečná reprodukce, která již ale poskytuje některé **zajímavé výsledky**
- zásuvný modul do překladače `gcc-4.5`

- dovést do praktičtější podoby teoretické výsledky publikované dříve
- zaměření na práci s daty uloženými v dynamických datových strukturách (např. kombinace s predikátovou abstrakcí)
- efektivní práce s programy s poli i dynamickými datovými strukturami
- automatické zjemňování abstrakce

- návrh algoritmů a datových struktur pro efektivní implementaci
- provést rozsáhlejší testování
- výsledkem bude **veřejně dostupný nástroj**, publikovatelný formou článku o nástroji a experimentech s ním (případně formou popisu použitých optimalizací)

Příklad 1 – různé programátorské chyby

```
1 #include <stdlib.h>
2
3 typedef struct list {
4     struct list *head;
5     struct list *next;
6 } list_t;
7
8 static void chain_item(list_t *list, list_t item) {
9     item = *list;
10    list->next = &item;
11 }
12
13 static void free_list(list_t *list) {
14     list = list->next;
15     while (list) {
16         list_t *next = list->next;
17         free(list);
18         list = next;
19     }
20 }
21
22 int main() {
23     list_t list = { .head = &list, .next = NULL };
24     list_t *item = (list_t *) malloc(sizeof item);
25     chain_item(&list, *item);
26     free_list(list.head);
27     return 0;
28 }
```

```
test-0028.c:24:13: error: amount of allocated memory
not accurate
test-0028.c:24:13: note: allocated: 4 bytes
test-0028.c:24:13: note: expected: 8 bytes
test-0028.c:25:15: error: dereference of NULL value
test-0028.c:16:17: error: dereference of
non-existing non-heap object
test-0028.c:26:14: note: from call of free_list()
test-0028.c:22:5: note: from call of main()
test-0028.c:17:13: error: attempt to free a non-heap
object, which does not exist anyhow
test-0028.c:26:14: note: from call of free_list()
test-0028.c:22:5: note: from call of main()
test-0028.c:24:13: warning: killing junk
```

Příklad 2 – OOM analýza

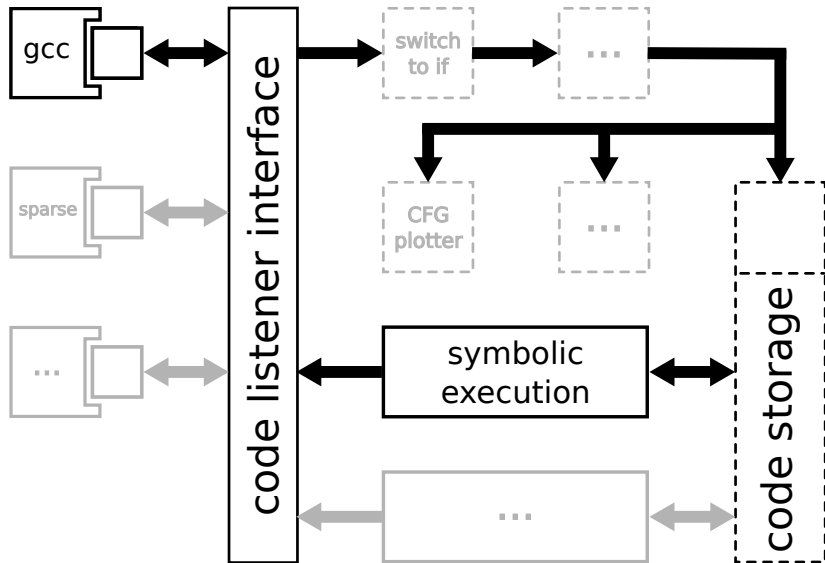
```
1 #include <stdlib.h>
2 #define NEW (item_t) malloc(sizeof(struct item))
3 typedef struct item {
4     struct item *lptr; struct item *rptr;
5 } *item_t;
6
7 item_t alloc_triple(void) {
8     item_t lptr, rptr, root = NULL;
9     if (NULL == (root = NEW)) return NULL;
10    if (NULL == (lptr = NEW)) goto out_of_memory;
11    if (NULL == (rptr = NEW)) goto out_of_memory;
12    root->lptr = lptr; root->rptr = rptr;
13    return root;
14 out_of_memory:
15    free(lptr);
16    free(rptr);
17    free(root);
18    return NULL;
19 }
20
21 int main() {
22     item_t item = alloc_triple();
23     if (item) {
24         free(item->lptr);
25         free(item->rptr);
26         free(item);
27     }
28     return 0;
29 }
```

```
test-0030.c:16:9: error: free() called on uninitialized
value
```

```
test-0030.c:22:12: note: from call of alloc_triple()
```

```
test-0030.c:21:5: note: from call of main()
```

Spolupráce s překladačem



- [Berdine *et al.*, 2005] Josh Berdine, Cristiano Calcagno, Peter W. O’hearn, and Queen Mary.
Symbolic execution with separation logic.
In *In APLAS*, pages 52–68. Springer, 2005.
- [Distefano *et al.*, 2006] Dino Distefano, Peter W. O’Hearn, and Hongseok Yang.
A local shape analysis based on separation logic.
In Holger Hermanns and Jens Palsberg, editors, *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302. Springer, 2006.
- [Reynolds, 2002] John Reynolds.
Separation logic: A logic for shared mutable data structures.
pages 55–74. IEEE Computer Society, 2002.

Děkuji za pozornost.

Linearizovaný kód

```
main():
    goto L1

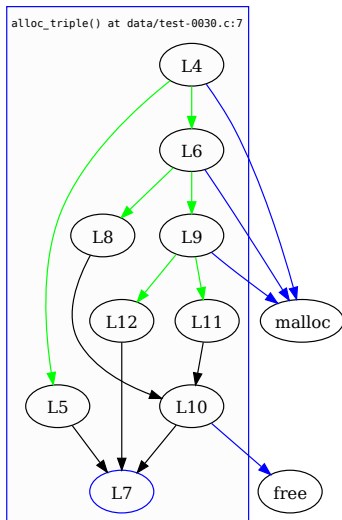
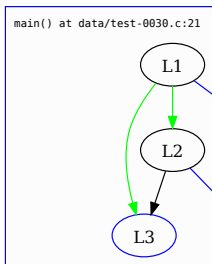
L1:
    %mF1:item := alloc_triple()
    %r1 := (%mF1:item != NULL)
    if (%r1)
        goto L2
    else
        goto L3

L2:
    %r2 := %mF1:item->[+0]lptr
    free(%r2)
    %r3 := %mF1:item->[+8]rptr
    free(%r3)
    free(%mF1:item)
    goto L3

L3:
    %r4 := 0
    ret %r4
```

Graf toku řízení (CFG)

data/test-0030.c



Symbolická hromada

